TURING 图灵程序设计丛书

Pearson

SQL Queries for Mere Mortals, Fourth Edition

SQL查询

从入门到实践（第4版）

[法] 约翰·L. 维斯卡斯（John L. Viescas）◎著　　袁国忠◎译

- 赢得世界赞誉的SQL经典著作
- 作者连续20年获微软MVP，凝结半个多世纪的数据库经验
- 亚马逊数据库图书榜单霸榜20年

中国工信出版集团　人民邮电出版社 POSTS & TELECOM PRESS

作者

约翰·L. 维斯卡斯（John L. Viescas）

拥有50多年经验的独立数据库咨询师。职业生涯之初，作为系统分析师，负责为IBM大型机系统设计数据库应用程序；后在Applied Data Research任职，负责IBM大型机数据库产品的研发和客户支持；1988年加入Tandem计算机公司，负责开发和实现市场营销数据库程序。

1993年自创公司，为全球各种规模的企业提供数据库咨询和培训服务。从1993年到2015年，创纪录地每年都被微软授予"最有价值专家"称号。

约翰写过多部关于数据库的著作，还为技术刊物写过许多文章，并在世界各地的会议和用户组会议上发表过演讲。

TURING 图灵程序设计丛书

SQL Queries for Mere Mortals, Fourth Edition

SQL查询

从入门到实践（第4版）

[法] 约翰 · L. 维斯卡斯（John L. Viescas）◎著　　袁国忠◎译

人民邮电出版社
北　京

图书在版编目（CIP）数据

SQL查询：从入门到实践 ：第4版 / （法）约翰·L.
维斯卡斯（John L. Viescas）著 ；袁国忠译. -- 北京 ：
人民邮电出版社，2020.4
（图灵程序设计丛书）
ISBN 978-7-115-53401-9

Ⅰ. ①S… Ⅱ. ①约… ②袁… Ⅲ. ①SQL语言－教材
Ⅳ. ①TP311.132.3

中国版本图书馆CIP数据核字(2020)第017926号

内 容 提 要

本书由多年从事 SQL 方面研究和咨询工作的资深专家编写，重点讲解 SQL 查询和数据操作的相关主题，包括关系型数据库和 SQL、SQL 基础、多表操作、汇总和分组数据，以及修改数据集等内容，针对编写 SQL 查询提供了轻松易懂的逐步指导，并包含上百个带有详细说明的例子。附录列出了所有 SQL 语句的语法图和示例数据库的结构等。

本书适合 SQL 开发人员以及经常需要与 SQL 打交道的程序员阅读。

◆ 著　　[法] 约翰·L. 维斯卡斯
译　　袁国忠
责任编辑　岳新欣
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：880×1230　1/16
印张：30.5
字数：1034千字　　2020年4月第1版
印数：1－3 500册　　2020年4月北京第1次印刷

著作权合同登记号　图字：01-2018-7601号

定价：149.00元

读者服务热线：(010)51095183转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版 权 声 明

对本书的赞誉

“优秀的图书让你知其然，卓越的图书让你知其所以然，本书无疑属于后者。要最大限度地挖掘数据库的潜力，必须将数据视为集合，而本书以简洁易懂的方式诠释了如何实际使用 SQL 及其背后的集合思维。”

——Ben Clothier，IT Impact 公司首席开发人员、《Access 2013 Web 编程》合著者、Microsoft Access MVP

“还记得七八年级学习图解句子的情形吗？现在公立学校不再教这些了，真遗憾。你可能不记得当时是如何图解句子的，但正是这种做法让你的写作水平提高了。John Viescas 肯定还记得这样的情形，因为他竟然将用自然语言阐述的问题转换成了 SQL。对任何数据库设计人员来说，这都是一部重要的著作，它举重若轻，将 E. F. Codd 撰写的关系型数据库设计论文中阐述的复杂集合论和一阶谓词逻辑细细道来，让任何人都很容易明白。要将 SQL 水平从初级提高到中级，一定要读本书——不管你购买了多少本其他的图书。”

——Arvin Meyer，MCP、MVP

“即便有了向导和代码生成器，数据库开发人员依然必须牢固地掌握结构化查询语言（SQL，用于与大多数数据库系统交流的标准语言）。本书以幽默而合乎逻辑的方式将乏味而晦涩难懂的主题讲得有声有色，并辅以大量相关示例，任何严肃认真的开发人员都应将其置于书架上。你肯定会经常翻阅它，因此它待在书架上的时间不会太长！”

——Doug Steele，Microsoft Access 开发人员、作者

“如果你需要处理数据，强烈建议你阅读这本书，它将让你轻松学会数据处理的一个最重要的方面——编写查询。查询是重要的数据选择、排序和报告工具，可弥补表结构的不足、满足新的报告需求、纳入新的数据源。本书语言清晰易懂，通过大量的示例演示了如何解决从简单到复杂的问题，并将概念应用于各种不同的场景。无论你是新手还是专家，本书都极具参考价值。”

——Teresa Hennig，Microsoft Access MVP、多部 Access 著作的第一作者

序

数据库语言 SQL 成为国际标准已有 30 年，SQL 数据库产品横空出世也已 30 年。在此期间，SQL 逐渐在存储、修改、检索和删除数据的语言中占据了统治地位，当今世界的数据（乃至经济）很大一部分是使用 SQL 数据库记录的。

作为一款功能极其强大的数据操作工具，SQL 已无处不在。它现身于高性能事务处理系统，隐藏在 Web 界面后面，甚至网络监视工具和垃圾邮件防火墙中都有它的身影。

当前，SQL 可直接执行，可嵌入到编程语言中，还可通过调用接口访问。它隐身于 GUI 开发工具、代码生成器和报告生成器中，但无论是在台前还是幕后，底层的查询都是 SQL，因此无论你是想理解现有的应用程序还是要创建新的应用程序，都必须熟悉 SQL。

本书循序渐进地介绍了如何编写 SQL 查询，语言清晰易懂，并且通过大量的示例和详细说明来帮助你掌握理解、修改和编写 SQL 查询所需的技能。

作为数据库咨询师，以及美国 SQL 标准委员会和国际 SQL 标准委员会的成员，我拥有丰富的 SQL 使用经验，因此可以负责任地说，本书作者不仅谙熟 SQL，而且擅长向他人讲述 SQL，这两种品质让本书成了一份极其宝贵的参考资料。

Keith W. Hare
JCC 咨询公司高级咨询师
美国 SQL 标准委员会（INCITS DM32.2）副主席
国际 SQL 标准委员会（ISO/IEC JTC1 SC32 WG3）召集人

导　　言

“语言从本质上说是一种交流工具，它表达的不是具体的事情，而是大家的共识。”

——托马斯·厄内斯特·休姆，《意度集》

学习检索和操作数据库中的信息通常是一项艰巨的任务，但只要搞明白了要回答的问题或要对数据库做的修改，这项任务完成起来就会很容易。搞明白问题后，就可将其翻译成数据库系统使用的语言——在大多数情况下是结构化查询语言（SQL）。你必须将问题转换为 SQL 语句，让数据库系统知道你要检索或修改哪些信息。SQL 为你提供了与数据库系统交流的手段。

在多年的数据库咨询生涯中，我发现很多人只需从数据库检索信息或执行简单的数据修改，这样的人远远多于数据库应用程序开发人员。但市面上没有只介绍这方面内容的图书，更别说从普通人的角度进行介绍的书了。诚然，介绍 SQL 的优秀图书很多，但大都以数据库编程和开发为目标。

有鉴于此，我认为该编写一本书来介绍如何正确而有效地查询数据库了。在我与好友 Michael J. Hernandez 的努力下，本书第 1 版于 2000 年出版。2008 年，我们又推出了第 2 版，主要介绍使用 SQL 修改数据库数据的基本方法。在 2014 年推出的第 3 版中，我们稍微涉足了更棘手的问题——即便是经验丰富的用户也会头痛的问题。您手里捧着的是第 4 版，它在棘手问题方面涉足更深，新增了对窗口函数和分组集（GROUPING SETS）的介绍。相比于其他 SQL 图书，本书的独特之处在于专注于 SQL 而不针对任何具体的数据库系统。这一版新增了数十个示例，并提供了 4 个版本的示例数据库：Microsoft Office Access、Microsoft SQL Server、流行的开源数据库系统 MySQL 和 PostgreSQL。阅读完本书后，你将具备检索和修改信息所需的技能。

前　　言

“我想，你是凡夫俗子，可能会犯错。”

——詹姆斯·雪利，*The Lady of Pleasure*

如果你经常使用计算机，很可能使用过结构化查询语言（Structured Query Language，SQL），只是没有意识到而已。SQL 是用于与大多数数据库系统交流的标准语言。每当你将数据导入电子表格或将数据合并到字处理程序时，很可能在以这样或那样的方式使用 SQL；每当你访问电子商务网站并下单购买图书、唱片、电影等各种商品时，你使用的网页后面的代码极可能在使用 SQL 访问数据库。如果你需要从使用 SQL 的数据库系统获取信息，可通过阅读本书来加深对这种语言的认识。

你是凡夫俗子吗

你可能会问：“谁是凡夫俗子，我吗？”答案并非那么简单。着手编写本书时，我自认为是数据库语言 SQL 方面的专家，但一路走来，我发现自己在很多方面也是凡夫俗子。我对 SQL 的某些具体实现了如指掌，但在研究众多商业数据库系统如何使用这门语言的过程中，我发现了很多纷繁复杂之处。因此，只要符合下面描述的任何一条，你便也是凡夫俗子。

- 如果你使用计算机程序来访问数据库系统中的信息，你很可能是凡夫俗子。使用应用程序内置的查询工具未能获得期望的信息时，你就需要研究底层的 SQL 语句，找出其中的原因。
- 如果你最近在使用桌面数据库应用程序，却在使用它定义和查询所需的数据时举步维艰，你就是凡夫俗子。
- 如果你是数据库程序员，需要跳出习惯性思维去解决一些复杂的问题，你就是凡夫俗子。
- 如果你是精通某种数据库系统的专家，但需要将该系统中的数据整合到另一种支持 SQL 的系统中，你就是凡夫俗子。

简而言之，只要你需要使用支持 SQL 的数据库系统，就适合阅读本书。如果你是数据库新手，刚知道可使用 SQL 来获取所需的数据，你将发现本书介绍了所有的基础知识。如果你是专家级用户，但需要解决复杂的问题或集成多个支持 SQL 的系统，那么本书能够帮助你深刻认识如何利用数据库语言 SQL 的复杂功能。

涵盖的内容

本书的所有内容都是基于国际标准化组织（ISO）制定的最新 SQL 数据库语言标准——SQL/Foundation（文档 ISO/IEC 9075-2:2016）编写的。流行的商业数据库系统当前大都实现了这个标准。该 ISO 文档也被美国国家标准学会（ANSI）采纳，因此这是一个货真价实的国际标准。本书介绍 SQL 时，不针对任何特定的数据库产品。

SQL 标准定义的功能可能比大部分商业数据库产品实现的功能更多，也可能更少。大多数数据库厂商还没有实现很多高级功能，但都支持该标准定义的核心功能。

我研究了众多流行的数据库产品，旨在确保你能将本书介绍的知识付诸应用。对于 SQL 语言的核心功能，如果我发现有些主流数据库产品不支持，我会指出这一点，并介绍 SQL 标准中的替代功能；对于 SQL 标准定义的重要功能，如果我发现支持的厂商不多，我会简单地介绍其语法，再提供替代方案。

本书分为如下六个部分。

- 第一部分（关系型数据库和 SQL）指出现代数据库系统基于严格的数学模型，简要地介绍数据库查询语言 SQL 的发展历程，并讨论一些确保数据库设计合理的简单规则。
- 第二部分（SQL 基础）介绍如何使用 SELECT 语句、如何编写表达式、如何使用 ORDER BY 子句进行排序，以及如何使用 WHERE 子句筛选数据。
- 第三部分（使用多个表）演示如何编写从多个表中获取数据的查询，包括如何使用内连接、外连接和 UNION 将表关联起来，以及如何使用子查询。
- 第四部分（数据汇总和分组）讨论如何获取汇总信息以及如何对其进行分组和筛选。在这一部分，你将学习 GROUP BY 和 HAVING 子句。
- 第五部分（修改数据集）讨论如何编写查询来修改表中的一系列行。在这一部分，你将学习如何使用 UPDATE、INSERT 和 DELETE 语句。
- 第六部分（解决棘手问题）介绍更复杂的问题。在这一部分，你将拓宽视野，了解如何解决复杂的否定型问题和多条件问题、如何使用 CASE 执行条件测试、如何跳出习惯性思维使用非连接表（笛卡儿积），还将学习如何在 GROUP BY 中使用其他关键字来生成小计和累积以及将输出数据划分为多个子集。

本书末尾有几个附录，列出了本书介绍过的所有 SQL 元素的语法图、示例数据库的结构、6 种主流数据库系统实现的日期和时间操作函数，以及可帮助你深入了解 SQL 的推荐读物。我针对 4 种数据库系统（Microsoft Access、Microsoft SQL Server、MySQL 和 PostgreSQL）提供了 5 个示例数据库。

未涉及的内容

虽然本书是基于 SQL: 2016 标准编写的，但并未涵盖该标准的方方面面。实际上，在未来的多年内，SQL: 2016 标准定义的很多功能不会在主流数据库系统中实现。本书的首要目标是帮助你牢固地掌握如何使用 SQL 编写查询。在本书中，我时不时会建议你参阅你使用的数据库系统的文档，了解当前介绍的功能是否可行。这并不意味着本书只介绍主流数据库系统都实现了的功能，但介绍有些系统没有实现或实现方式不同的功能时，我会尽可能提醒你。

如果数据库设计存在缺陷，将难以编写使用多个表的复杂查询。我专辟一章介绍了数据库设计，旨在让你知道在什么情况下你可能会遇到麻烦，但该章只介绍了一些基本原则。详细讨论数据库设计原则以及如何在特定数据库系统中实现设计，不在本书的范围之内。

另外，以效率最高的方式解决问题不是本书的目标所在。阅读本书的很多章节时，你将发现对于特定的问题，我提供了多种解决方法。在有些情况下，以某种方式编写查询在任何数据库系统中都可能遭遇性能问题，此时我会给予提醒。但每种数据库系统都有其优势和劣势，学习基础知识后，你就可深入研究你使用的数据库系统，学习如何编写出运行效率最高的查询。

导读

编写本书时，我假定你会按顺序阅读，每章都建立在介绍过的概念的基础之上。但即便你直接从本书中间开始阅读，也不至于迷失方向。例如，如果你熟悉 SELECT 语句中的基本子句，并想更深入地学习连接，可直接阅读第 7~9 章。

很多章中提供了大量的示例、解决方案和结果集。建议你研究一些示例，对相关的技术有更深入的认识后，再尝试在不看我提供的解决方案的情况下完成一些练习。

请注意，在查询返回的结果集包含数十行时，我只列出前几行，以便让你对结果集的样子有大致的了解。然而，在你使用的数据库系统中，你看到的结果可能不同，因为支持 SQL 的数据库系统都有优化器，优化器负责找出执行查询的最快方式。另外，你使用的数据库系统返回的前几行可能与书中显示的前几行并不完全相同——除非在查询中使用了 ORDER BY 子句来指定以什么样的顺序返回各行。

很多章的末尾都提供了大量的练习，让你有机会将学到的知识付诸应用。不用担心，解决方案都可在你从本书配套网站下载的示例数据库中找到。对于完成起来有点棘手的练习，我还给出了提示。

在本书的最后，附录 A 列出了完整的 SQL 标准语法图，无论你要了解本书介绍的哪种 SQL 技术，这些语法图都极具参考价值。附录 B 列出了示例数据库的结构，你可参考它们来设计自己的数据库。

语法图解读

本书包含大量语法图，演示了使用 SQL 时将涉及的语句、术语和词语的正确语法。每个语法图都清楚地说明了相应 SQL 元素的总体结构，你可以以这些语法图为模板编写 SQL 语句，还可以使用它们来帮助理解示例。

这些语法图涉及一系列核心元素，这些元素分为两大类：**语句**和**语句项**。语句是主要的 SQL 操作，如 SELECT 语句；而语句项是语句的组成部分，如**值表达式**、**查找条件**、**谓词**（别担心，本书后面将介绍所有这些术语）。语句语法图和语句项语法图的唯一差别是主语法线的端点不同，这种差别旨在让你清楚地知道查看的是整条语句的语法图，还是可能在语句中使用的语句项的语法图。图 I-1 显示了这两种语法图中语法线的起点和终点。除这种差别外，这两种语法图使用的元素是相同的。图 I-2 显示了这两种语法图的示例，接下来将对每个语法图元素做简单说明。

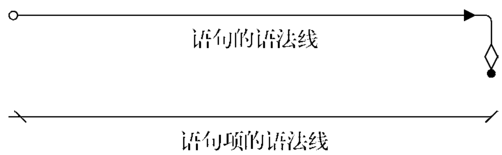


图 I-1　语句和语句项的语法线端点

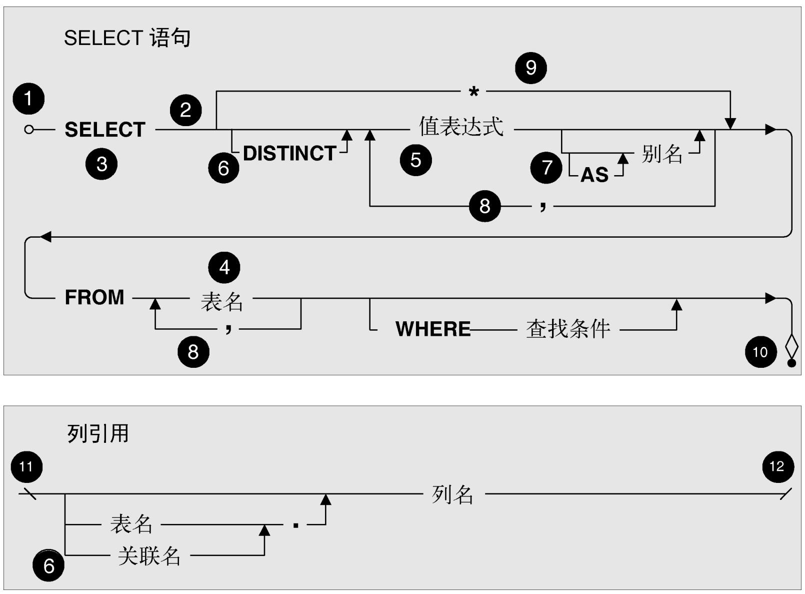


图 I-2　语句语法图和语句项语法图示例

❶ 语句的起点：表示语句的主语法线的起始位置。位于主语法线内的元素都是必不可少的，而位于主语法线下方的元素都是可选的。

❷ 主语法线：指出了语句或语句项的必不可少和可选元素的排列顺序。编写语句或语句项时，必须按从左到右（或箭头方向指定）的顺序提供元素。

❸ 关键字：语句或语句项的语法中必不可少的单词。在语法图中，关键字全大写并以粗体显示（在数据库程序

中编写语句时，并非一定要以全大写方式指定关键字，但这样做可提升语句的可读性）。

❹ **按字面意思解读的项**：告诉你需要在语句中提供什么样的值。

❺ **语句项**：表示某种操作，它返回将在语句中使用的最终值。对于你需要知道的所有语句项，本书后面都将介绍并提供其语法图。

❻ **可选元素**：位于主语法线下方的元素。可选元素可能是语句、关键字、语句项或字面量值，为清晰起见，它们都有自己的语法线。在有些情况下，可将选项指定为一系列用逗号分隔的值（参见❽）。另外，有些可选元素包含一系列可选的子元素（参见❼）。通常，沿可选元素的语法线从左往右读，就像主语法线一样，但在有箭头的情况下，务必沿箭头方向读，这样就不会有麻烦。请注意，对于有些选项，可将其指定为多个值，在这种情况下，箭头是从右到左的；但输入这些值后，顺序将恢复到正常的从左到右。所幸所有可选元素的工作原理都相同，因此你在本书后面学习一个可选元素的用法后，便知道如何使用语法图中的任何可选元素。

❼ **可选子元素**：位于可选元素下方的一个或多个元素。可选子元素让你能够通过微调语句来解决更复杂的问题。

❽ **选项列表分隔符**：指出可将当前选项指定为多个值，但必须用逗号分隔。

❾ **替代选项**：可用来替代一个或多个可选元素的关键字或语句项，其语法线绕过被替代可选元素的语法线。

❿ **语句的终点**：语句的主语法线的终点。

⓫ **语句项的起点**：语句项的主语法线的起点。

⓬ **语句项的终点**：语句项的主语法线的终点。

熟悉这些元素后，你就能读懂本书所有的语法图。在需要对语法图做进一步说明时，我会提供必要的信息，让你能够轻松读懂。为了帮助你更好地理解语法图，下面给出了我根据图 I-2 编写的一条 SELECT 语句。

```
SELECT FirstName, LastName, City, DOB AS DateOfBirth
FROM Students
WHERE City = 'El Paso'
```

这条 SELECT 语句从 Students 表中获取 4 列，这些列和表分别是在 SELECT 和 FROM 子句中指定的。如果你沿主语法线从左往右前行，将发现必须至少指定一个**值表达式**。值表达式可以是列名、使用列名编写的表达式或要显示的常量（字面量）。通过使用值表达式的**选项列表分隔符**（逗号），可指定任意数量的列，这就是前面能够使用 Students 表中 4 个列名的原因。我担心查看者看到这条 SELECT 语句返回的信息时，不知道 DOB 是什么意思，因此我使用值表达式的子选项 AS 给 DOB 列指定了一个**别名**。最后，我使用 WHERE 子句确保这条 SELECT 语句只返回居住在 El Paso 的学生（如果你现在还不太明白，不用担心，本书后面将详细介绍这些内容）。

附录 A 列出了所有的语法图，展示了本书讨论的所有语句和语句项的完整语法。如果你在阅读各章时参阅这些语法图，将发现附录 A 和章节中有些相应的语法图存在细微的差别，这是因为章节中的语法图是简化版。利用这些简化版语法图，我可以更轻松地对复杂的语句和语句项做出解释，并在必要时将重点放在特定的元素上。别担心，等你阅读完本书后，附录 A 中所有的语法图理解起来都会很容易。

示例数据库

本书的示例查询中使用了 5 个示例数据库。要下载它们，可访问本书配套网站 www.informit.com/title/9780134858333 并单击标签 Downloads。[①]这些数据库的结构见附录 B。

(1) Sales Orders：自行车和配饰商店使用的典型订单数据库。（介绍数据库的图书都得至少提供一个订单示例，不是吗？）

(2) Entertainment Agency：这个数据库用于管理演唱组合、经纪人、顾客和演出合约。你可使用一个设计方式类似的数据库来管理活动预订和酒店预订。

(3) School Scheduling：可在高中或社区大学使用设计方式类似的数据库来管理学生注册。这个数据库不仅记录课程注册情况，而且记录讲授每门课程的教员以及学生成绩。

---

① 也可以从 ituring.cn/book/2628 下载。——编者注

(4) Bowling League：这个数据库记录保龄球队、各队的队员及其参加的比赛和结果。

(5) Recipes：可使用这个数据库来保存和管理喜欢的菜品，我还在其中添加了一些你可能想试着做一做的菜品。

这 5 个示例数据库都有 4 种版本。

- 考虑到桌面数据库系统 Microsoft Office Access 如此流行，我使用 Microsoft Access 2007（12.0 版）创建了一组数据库（文件扩展名为.accdb）。我之所以选择 Microsoft Access 2007，是因为它大致支持最新的 ISO/IEC SQL 标准，同时可使用 Access 2007、2010、2013 或更高的版本打开这种格式的数据库文件。这些文件位于文件夹 MSAccess 中（我使用 Microsoft Access 2016 测试了所有的示例查询）。
- 第二种格式的数据库文件（文件扩展名为.mdf）是使用 Microsoft SQL Server 2016 Express Edition 创建的，这些文件位于文件夹 MSSQLServer 中，可在 Microsoft SQL Server 2016 或更高版本中附加它们。我还提供了 SQL 命令文件（文件扩展名为.sql），可在 Microsoft SQL Server 中使用它们从头创建示例数据库。这些文件位于文件夹 SQLScripts 中。
- 第三组数据库是使用流行的开源数据库系统 MySQL 5.7.18 社区版创建的。你可使用文件夹 SQLScripts 中的脚本（文件扩展名为.sql），在你自己的 MySQL 数据文件夹中创建数据库、加载数据以及创建示例视图和存储过程。
- 第四组数据库是使用流行的 PostgreSQL 9.6.3 创建的。与 MySQL 一样，可使用文件夹 SQLScripts 中的脚本（文件扩展名为.sql），在你自己的 MySQL 数据文件夹中创建数据库、加载数据以及创建示例视图和函数。

要安装示例文件，请参阅从 www.informit.com/title/9780134858333 下载的压缩包中的 ReadMe.txt 文件。

❖ **说明**　虽然我小心谨慎，在 SQL 示例脚本中使用的是命令 CREATE TABLE、CREATE INDEX、CREATE CONSTRAINT 和 INSERT 等最常见、最简单的语法，但你（或数据库管理员）依然可能需要稍微修改一下这些文件，以便在你使用的数据库系统中运行。如果你使用的是远程服务器上的数据库系统，可能需要请数据库管理员授予必要的权限，以便使用我提供的 SQL 命令来创建数据库。

第二、三、四、六部分介绍的是 SELECT 语句，在这些部分的章节中，所有的示例查询和解决方案都在相应数据库的“示例”版（如 SalesOrdersExample、EntertainmentAgencyExample）中。第五部分的示例修改了数据，因此我创建了每个示例数据库的修改版（如 SalesOrdersModify、EntertainmentAgencyModify）。这种版本的示例数据库包含额外的列和表，让我能够演示 UPDATE、INSERT 和 DELETE 查询的某些功能。

❖ **警告**　无论是介绍概念还是编写示例语句，我使用的都是 ISO 标准 SQL。在很多情况下，可直接使用这种 SQL 来创建示例视图、函数和存储过程，但在有些情况下，我必须修改示例 SQL，以使其能够在目标数据库系统中正确运行。例如，编写日期表达式或执行日期计算时，我使用了目标数据库系统支持的相关函数（附录 C 列出了 6 种主流数据库系统支持的所有时间和日期函数）。

另外，虽然我使用的脚本与书中示例显示的大致相同，但在保存视图、函数和存储过程前，Microsoft SQL Server、MySQL 和 PostgreSQL 会出于优化目的修改 SQL。如果你使用 SQL Server Management Studio 中的 Design 界面、MySQL Workbench 中的 Alter 或 PostgreSQL pgAdmin 来编辑视图或存储过程，看到的将是保存在数据库中的 SQL，它们可能与我定义视图或存储过程时使用的 SQL 有天壤之别。心存疑惑时，请务必参阅配套脚本文件，看看我使用的 SQL 是什么样的。

“沿黄砖路往前走。”

——《绿野仙踪》中蒙奇金人送给多萝西的话

看完了前言，可以马上学习 SQL 了吧？也许吧。

听我的建议，阅读第 4 章前，先阅读前 3 章。通过阅读第 1 章，你将大致了解关系型数据库是如何面世的，以及它是如何成为当今业界使用最广泛的数据库的。我希望这能够让你对你使用的数据库系统有些认识。第 2 章介绍如何

微调数据库结构，让数据可靠而准确。如果数据库结构设计得不合理，有些 SQL 语句将难以编写，因此建议你仔细阅读这一章。

第 3 章简直就是“黄砖路”的起点。通过阅读该章，你将知道 SQL 的起源以及它是如何成为现在这样的。你还将知道一些催生 SQL 的人和公司，以及 SQL 为何有如此多的变种。最后，你将知道 SQL 是如何成为国家和国际标准的，以及它在未来几年的发展方向。

阅读完这 3 章后，便为前往“奥兹国”做好了充分准备，只需沿我在后续各章铺设好的路前行即可。阅读完本书后，你就会发现向导找到了——这个向导就是你自己。

致　　谢

像本书这样的图书得以付梓，是众人合作的结果。本书的出版离不开编辑、同事、朋友、亲戚的支持和宝贵建议。在本书编写过程中，他们不断地鼓励我、让我专心致志并给我动力。

首先，我要感谢策划编辑 Trina MacDonald 帮助我签订编写本书第 4 版的合同。感谢开发编辑 Rick Kughen 一路上的指导和督促。感谢制作团队，你们非常出色！还要感谢技术编辑 Doug Steele 和数据库领域的朋友 Ben Clothier。感谢你们所有人投入时间和精力，帮助我将这本大部头的 SQL 查询图书推向市场。

最后，要特别感谢 Keith Hare 为本书作序。身为国际 SQL 标准委员会召集人，Keith 是一位出类拔萃的 SQL 专家，其 SQL 专业知识精深而广博，我对他敬佩有加。能够在本书开头将其想法和评论呈现给读者，我深感荣幸。

目　　录

第一部分　关系型数据库和SQL

第二部分　SQL 基础

第三部分 使用多个表

第四部分 数据汇总和分组

第五部分 修改数据集

第六部分 解决棘手问题

第一部分

关系型数据库和 SQL

第 1 章

何谓关系型数据库

1

“知识是从无知中整理和分类出来的很小一部分。”

——安布罗斯·比尔斯

本章涵盖如下主题：

- 数据库类型
- 关系模型简史
- 关系型数据库剖析
- 学习关系型数据库有何好处
- 小结

学习 SQL 前，先得搞明白 SQL 支持的数据库结构背后的逻辑。本章介绍最初设计关系型数据库的原因、它的结构以及为何要使用它。这些信息可帮助你理解 SQL 到底是做什么的以及如何最大限度地利用它。

1.1　数据库类型

数据库是什么呢？你可能知道，数据库是组织有序的数据集，用于模拟某种类型的组织或组织流程。当你出于某种目的以组织有序的方式收集并存储数据时，便创建了数据库，至于你是使用纸张还是应用程序来收集并存储数据，这无关紧要。在这里的讨论中，我假定你使用应用程序来收集并维护数据。

数据库管理中使用的数据库分为两大类：**操作型数据库**和**分析型数据库**。

对当今世界的很多公司、组织和机构来说，操作型数据库是其支柱。这种数据库主要用于日复一日地收集、修改和维护数据，其中存储的数据是**动态**的，即不断变化，始终反映最新的状态。零售商、制造商、医院、诊所、出版社等组织使用操作型数据库，因为它们的数据在不断地变化。

相反，分析型数据库存储并跟踪与时间相关的历史数据。这是一项宝贵的资产，让你能够跟踪趋势、查看长期的统计数据、做出战略或战术性业务预测。这种数据库存储的数据是**静态**的，即根本不会或很少发生变化，虽然可能随时间的流逝不断添加新数据。从分析型数据库中收集的信息反映的是特定时点的状态，通常不是最新的。使用分析型数据库的组织包括化学实验室、地质公司、市场分析公司等。请注意，分析型数据库中的数据通常是从操作型数据库中提取出来的，例如，可能在分析型数据库中汇总并存储以往的月度销售数据。

1.2　关系模型简史

数据库模型有多种，其中有些只能在遗留系统中看到，如层次模型和网络模型，而有些被人们广泛接受，如关系模型。在其他图书中你可能还见过其他模型，如对象模型、对象–关系模型、联机分析处理（OLAP）模型。实际上，SQL 标准定义了支持这些模型的扩展，而有些商用数据库系统实现了这些扩展，但这里只讨论关系模型以及国际 SQL 标准的核心。

1.2.1　起源

关系型数据库起源于1969年，现已成为数据库管理中使用最广泛的数据库模型。20世纪60年代末，关系模型之父 Edgar F. Codd 博士（1923—2003）在 IBM 担任研究员，致力于寻找处理海量数据的新方法。他对当时的数据库模型和数据库产品很不满意，于是萌生了利用数学原理来解决各种问题的想法。作为数学家，他深信自己完全能够利用数学分支来解决诸如数据冗余、数据完整性不佳以及数据库结构过度依赖于物理实现等问题。

1970年6月，Codd 博士发表了具有划时代意义的文章"A Relational Model of Data for Large Shared Databanks"①，正式提出了关系模型。这个模型以两个数学分支为基础——集合论和一阶谓词逻辑；实际上，这个模型的名称就源于集合论术语**关系**（relation）。（一个普遍的误解是，"关系模型"这个名称源自关系型数据库中的表彼此相关这一事实，但实际上，这里的"关系"指的是关系型数据库系统中的表。知道真相后，你今晚就能睡个安稳觉了！）所幸你无须了解集合论和一阶谓词逻辑的细节就能设计和使用关系型数据库。只要采用良好的数据库设计方法，如 Mike Hernandez 在其著作 *Database Design for Mere Mortals* 第3版中介绍的方法，就可设计出合理而高效的数据库结构，并将其用于收集和维护任何数据。当然，要解决更复杂的问题，还是需要对谓词和集合论有大致的了解。谓词其实就是筛选器，这将在第6章介绍，而第7章将简要地介绍集合论。

1.2.2　关系型数据库系统

关系型数据库管理系统（RDBMS）是用于创建、维护、修改和操作关系型数据库的应用程序软件。很多 RDBMS 程序都提供了相关的工具，让你能够创建与数据库中存储的数据交互的最终用户应用程序。RDBMS 程序自面世后不断发展；随着硬件技术和操作环境的发展，这些程序的功能日益齐备而强大。

关系型数据库刚面世时，编写的 RDBMS 用于在大型机上运行。20世纪70年代初，有两个 RDBMS 程序很流行，分别是 IBM 加州圣何塞研究实验室开发的 System R，以及加州大学伯克利分校开发的 Interactive Graphics Retrieval System（INGRES）。这两个程序为关系模型的推广做出了重大贡献。

随着关系型数据库的优点变得广为人知，很多公司决定慢慢地从层次和网络数据库模型转向关系型数据库模型，对卓越的大型机 RDBMS 程序的需求与日俱增。20世纪80年代，Oracle 和 IBM 等公司推出了各种商用大型机 RDBMS 程序。

20世纪80年代初期和中期，随着个人计算机（PC）的崛起，基于 PC 的 RDBMS 程序应运而生。这类早期产品中，有些不过是基于文件的简单数据库管理系统，如 Ashton-Tate 和 Fox Software 等公司推出的产品。等到 Microrim 和 Ansa Software 等公司推出相关的产品后，真正意义上的基于 PC 的 RDBMS 程序才出现。这些公司帮助推广了数据库管理的理念和潜能，使其从大型机占统治地位的信息系统部门走向普通用户的桌面。

20世纪80年代末到90年代初，使用数据库的用户越来越多，分享数据的需求变得不言而喻。让多个用户使用同一个中央数据库的理念看起来前途无量，这显然将简化数据管理和数据库安全工作。为了满足这种需求，Microsoft 和 Oracle 等数据库厂商开发了**客户端/服务器型** RDBMS 程序。

多年来，数据库的使用方式发生了翻天覆地的变化，很多组织逐渐意识到，可从其存储在各种关系型和非关系型数据库中的数据中提取大量宝贵的信息。有鉴于此，他们提出了这样的问题：能否对数据进行挖掘，从中提取出宝贵的分析信息，作为重要商务决策的依据？他们还提出了这样的问题：能否将数据整合成可靠的知识库？实际上，这些问题都难以回答。

IBM 提出了**数据仓库**的概念，旨在让组织能够访问存储在大量非关系型数据库中的数据。由于性能问题和复杂性较高，IBM 最初为实现数据仓库所做的尝试以失败告终，直到20世纪90年代，才出现了可行而实用的数据仓库实现。William H.（Bill）Inmon 被普遍认为是数据仓库之父，他极力提倡这种技术，并为其发展做出了巨大贡献。多年来，随着公司越来越多地利用存储在其数据库中的数据，数据仓库更为司空见惯了。

随着互联网的出现，组织使用数据库的方式发生了巨变。很多公司和企业都利用万维网来拓展消费群，它们与消

---

① *Communications of the ACM*，1970年6月刊第377~387页。

费者分享以及从消费者那里收集的数据大都存储在数据库中。开发人员普遍使用可扩展的标记语言（XML）来收集并合并来自各种关系型和非关系型数据库的数据。

为了让客户端能够在云端创建数据库并存储数据，各个厂商做出了巨大的努力。云端是一个与客户端完全分离的位置，其背后的理念是，客户端可随时随地通过互联网访问云端数据库中的数据。例如，为了支持云端数据库管理，在最新的几个 Microsoft Access 版本中，Microsoft 都将重点放在了将数据从桌面设备迁移到云服务器上。过去几年，联网设备的使用日益普遍，这种环境中的数据库管理系统将发生有趣的变化。

1.3　关系型数据库剖析

根据关系模型的定义，在关系型数据库中，数据存储在**关系**（就是用户看到的表）中。每个关系（relation）都由**元组**（记录或行）和**属性**（字段或列）组成。关系型数据库还有其他几个特征，这些都将在本节讨论。

1.3.1　表

表是数据库中最重要的元素，每个表都表示一个特定的客体。在表中，行和列的逻辑顺序无关紧要。每个表至少包含一列，该列被称为**主键**，它唯一地标识了表中的每一行。例如，在图 1-1 所示的 Customers 表中，主键为 CustomerID。实际上，正是由于表的这两个特征，关系型数据库中的数据才能够独立于其在计算机中的存储方式。对用户来说，这是天大的好消息，因为这意味着无须知道行的物理位置就能检索其中的数据。

表表示的客体可以是**物件**，也可以是**事件**。表示的客体为物件时，表表示的是看得见、摸得着的东西，如人物、地点或物体。无论是哪种类型的物件，都可将其特征存储为数据，再以无数种方式对这些数据进行处理。驾驶员、产品、机器、学生、建筑物和设备都是可用表表示的物件。图 1-1 是一个最常见的表示物件的表。

Customers

CustomerID	FirstName	LastName	StreetAddress	City	State	ZipCode
1010	Angel	Kennedy	667 Red River Road	Austin	TX	78710
1011	Alaina	Hallmark	Route 2, Box 203B	Woodinville	WA	98072
1012	Liz	Keyser	13920 S.E. 40th Street	Bellevue	WA	98006
1013	Rachel	Patterson	2114 Longview Lane	San Diego	CA	92199
1014	Sam	Abolrous	611 Alpine Drive	Palm Springs	CA	92263
1015	Darren	Gehring	2601 Seaview Lane	Chico	CA	95926

行

列

图 1-1　一个示例表

表示的客体为事件时，表表示在特定时点发生的事件。事件有你想要记录的特征，你可将这些特征存储为数据，然后像处理表示物件的表一样处理这些信息。司法听证会、基金分配情况、实验结果和地质勘查都是你可能想记录的事件。在销售订单数据库中，订单可视为物件（表示订单的纸张），也可时视为事件（发货）。图 1-2 中的表表示每个人都经历过的事件——看病。

Patient Visit

PatientID	VisitDate	VisitTime	Physician	BloodPressure	Temperature
92001	2006-05-01	10:30	Ehrlich	120 / 80	98.8
96105	2006-05-02	11:00	Hallmark	160 / 90	99.1
96203	2006-05-02	14:00	Hallmark	110 / 75	99.3
97002	2006-05-01	13:00	Hallmark	112 / 74	97.5
98003	2006-05-02	9:30	Fournier	120 / 82	98.6
99014	2006-05-02	9:30	Fournier	120 / 80	98.8

图 1-2　一个表示事件的表

1.3.2 列

列是数据库中最小的元素，表示其所属表表示的客体的一个特征。列是用于存储数据的元素，你可检索列中的数据，再以任何想象得到的方式呈现它。别忘了，从数据中获得的信息的质量取决于你花了多少时间来确保列本身的结构完整性和数据完整性。不要低估列的重要性。

在设计合理的数据库中，每列都只包含一个值，而列名指出了这个值的类型。这使得在列中输入数据简单而直观。例如，看到列名 FirstName、LastName、City、State 和 ZipCode，你就知道该在这些列中输入什么样的值。另外，按州对数据排序以及查找所有姓氏为 Viescas 的人都非常容易。

1.3.3 行

表中的每行都表示一个独一无二的客体，它包含表中的所有列，不管这些列是否包含值。由于表的定义方式，数据库中的每行都由其主键列中独一无二的值标识。

例如，在图 1-1 中，每行都表示一个独一无二的顾客，而数据库中的每个顾客都由 CustomerID 列标识。每行都包含表中的所有列，其中每列都描述了相应顾客的某个方面。行是理解表间关系的关键，因为你需要知道一个表中的行是如何关联到另一个表中的行的。

1.3.4 键

键是表中扮演特殊角色的列。键的类型决定了它在表中扮演什么样的角色。虽然表可包含多种类型的键，但本书只讨论两种最重要的键：**主键**和**外键**。

主键由一列或多列组成，唯一地标识了表中的每一行（主键由多列组成时称为**复合主键**）。主键是最重要的，原因有两个：其**值**标识了数据库中的**特定行**；其包含的**列**标识了数据库中的**特定表**。主键还实现了表级完整性，并帮助建立与其他表的关系。数据库中的每个表都有主键。

在图 1-3 中，AgentID 列是一个典型的主键，它唯一地标识了 Agents 表中的每一行，并通过确保没有重复的行来实现表级完整性。它还被用来在 Agents 表和数据库中的其他表（如 Entertainers 表）之间建立关系。

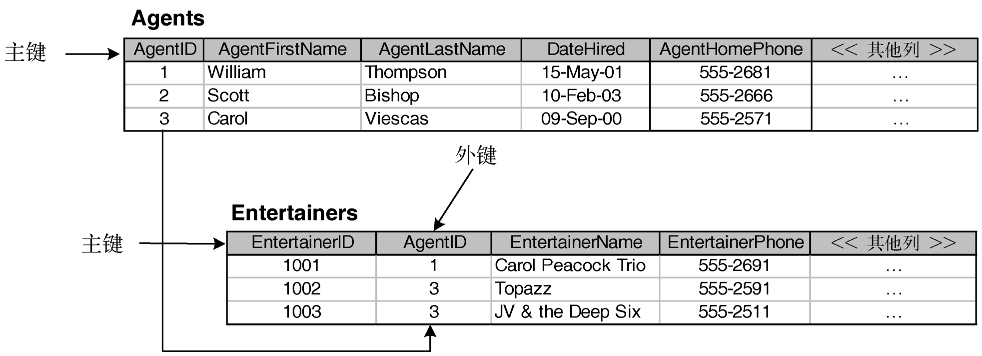


图 1-3 主键和外键

确定两个表彼此相关后，为了在它们之间建立关系，通常复制第一个表的主键，并将其插入到第二个表中。在第二个表中，这个主键被称为**外键**。（为何称为外键呢？因为第二个表有自己的主键，而你引入的第一个表的主键原本并不在第二个表中。）

图 1-3 还很好地演示了外键。在这个示例中，AgentID 是 Agents 表的主键，它还是 Entertainers 表的外键。如你所见，Entertainers 表已经有一个主键——EntertainerID。在这个示例中，AgentID 列将 Agents 表和 Entertainers 表关联起来了。

外键很重要，这不仅是因为它被用来在两个表之间建立关系，还因为它可确保关系级完整性。后者意味着两个表

中的行将被正确地关联起来，因为外键的值必须与其引用的主键的值相同。外键还有助于避免出现致命的“孤行”。一个典型的孤行示例是，订单行没有与之对应的顾客。如果不知道订单是谁下的，就无法对其进行处理，也就无法开具发货单，进而导致订单无法进入季度销售额统计中。

1.3.5　视图

视图是一个虚拟表，包含数据库中一个或多个表的列。这些构成视图的表被称为**基础表**（base table）。在关系模型中，视图之所以是虚拟的，是因为其中的数据来自基础表，视图本身并没有存储独立的数据。实际上，对于视图，数据库中只存储了其结构信息。

视图让你能够从不同的视角查看数据，从而在数据处理方面提供了极大的灵活性。你可以以不同的方式创建视图，而基于多个相关表的视图很有用。例如，你可以创建一个汇总信息的视图，其中列出西雅图市区每位木匠的工作总时间，还可以创建一个根据特定列对数据进行分组的视图，如列出指定地区各州每个城市的雇员总数。图 1-4 是一个典型的视图。

Customers

CustomerID	CustFirstName	CustLastName	CustPhone	<< 其他列 >>
10001	Doris	Hartwig	555-2671	…
10002	Deb	Waldal	555-2496	…
10003	Peter	Brehm	555-2501	…
<< 其他行 >>				

Engagements

EngagementNumber	CustomerID	StartDate	EndDate	StartTime	<< 其他列 >>
3	10001	2016-09-10	2016-09-15	13:00	…
13	10003	2016-09-17	2016-09-20	20:00	…
14	10001	2016-09-24	2016-09-29	16:00	…
17	10002	2016-09-29	2016-10-02	18:00	…

<< 其他行 >>

Customer_Engagements（视图）

EngagementNumber	CustFirstName	CustLastName	StartDate	EndDate
3	Doris	Hartwig	2016-09-10	2016-09-15
13	Peter	Brehm	2016-09-17	2016-09-20
14	Doris	Hartwig	2016-09-24	2016-09-29
17	Deb	Waldal	2016-09-29	2016-10-02
<< 其他行 >>				

图 1-4　一个示例视图

在很多 RDBMS 程序中，视图都是以**存储的查询**（简称为**查询**）的方式实现的。在大多数情况下，查询拥有视图的所有特征，因此查询与视图的唯一不同是名称不同。（我常常想，市场营销人员是否与此有关。）必须指出的是，有些厂商将视图直接称为查询。不管你使用的 RDBMS 程序将视图称为什么，你肯定都会用到它。

本书虽然名为“SQL 查询：从入门到实践”，但重点是介绍如何创建视图。正如第 2 章将介绍的，设计关系型数据库的正确方式是，对数据进行拆分，让每个表对应一个客体或事件。然而，在大多数情况下，你要获取的是有关客体或事件的信息，如各位顾客所下的订单或者各位教员讲授的课程。为此，你需要创建视图，因此需要知道如何使用 SQL 来创建视图。

1.3.6　关系

如果能够以某种方式将一个表中的行关联到另一个表中的行，就说这两个表之间存在关系。以什么样的方式建立

关系取决于关系的类型。表之间可存在的关系有三种：一对一、一对多和多对多。要搞懂视图的工作原理以及如何设计和使用多表 SQL 查询（这将在本书第三部分介绍），必须搞明白关系。

1. 一对一

如果有两个表，第一个表中的每行都**只**关联到第二个表中的一行，且第二个表中的每行都**只**关联到第一个表中的一行，这两个表的关系就是**一对一**的。在这种关系中，其中一个表被称为**主表**，而另一个表被称为**从属表**（secondary table）。为了建立这种关系，你将主表的主键作为外键插入到从属表中。这是一种特殊的关系，因为几乎在所有情况下，该外键同时也是从属表的主键。

图 1-5 演示了一个典型的一对一关系，其中 Agents 为主表，而 Compensation 为从属表。Agents 表中的每行都只关联到 Compensation 表中的一行，而 Compensation 表中的每行也都只关联到 Agents 表中的一行。请注意，AgentID 实际上是这两个表的主键，它还是从属表的外键。

Agents

AgentID	AgentFirstName	AgentLastName	DateOfHire	AgentHomePhone	<< 其他列 >>
1	William	Thompson	1997-05-15	555-2681	…
2	Scott	Bishop	1998-02-05	555-2666	…
3	Carol	Viescas	1997-11-19	555-2571	…

Compensation

AgentID	Salary	CommissionRate	<< 其他列 >>
1	$35,000.00	4.00%	…
2	$27,000.00	4.00%	…
3	$30,000.00	5.00%	…

图 1-5　一对一关系示例

在这种关系中，选择将哪个表作为主表呢？这取决于哪个表可包含在另一个表中没有匹配行的行。在一对一关系中，在从属表中不能添加在主表中没有匹配行的行。例如，雇用新的经纪人时，报酬可能还未确定。你需要能够在没有相应报酬行的情况下定义经纪人，因此 Agents 为主表。另外，给不存在的经纪人定义报酬行毫无意义，因此 Compensation 显然必须为从属表。一对一关系不那么常见，通常仅在出于保密的目的而将一个表一分为二时才会出现。

2. 一对多

两个表存在一对多关系时，其中一个表中的一行可能关联到另一个表中的**多行**，但后者中的一行只关联到前者中的**一行**。要建立这种关系，可将位于“一”端的表中的主键作为外键插入到位于“多”端的表中。

图 1-6 演示了一个典型的一对多关系。在这个示例中，Entertainers 表中的一行可能关联到 Engagements 表中的多行，但 Engagements 表中的一行只关联到 Entertainers 表中的一行。你可能猜到了，在 Engagements 表中，EntertainerID 为外键。

Entertainers

EntertainerID	EntertainerName	EntertainerPhone	<< 其他列 >>
1001	Carol Peacock Trio	555-2691	…
1002	Topazz	555-2591	…
1003	JV & the Deep Six	555-2511	…

Engagements

EngagementID	EntertainerID	CustomerID	StartDate	EndDate	<< 其他列 >>
5	1003	10006	2007-09-11	2007-09-14	…
7	1002	10004	2007-09-11	2007-09-18	…
10	1003	10005	2007-09-17	2007-09-26	…
12	1001	10014	2007-09-18	2007-09-26	…

图 1-6　一对多关系示例

3. 多对多

在两个表中，如果任何一个表中的一行都可能关联到另一个表中的**多行**，则这两个表之间存在多对多关系。为了正确地建立这种关系，必须创建**链接表**（linking table）。链接表让你能够轻松地将一个表中的行关联到另一个表中的行，还可避免添加、删除或修改相关的数据时出现问题。为了定义链接表，可复制多对多关系中每个表中的主键，并根据它们来定义链接表的结构。这些列有两个作用：它们一起构成了链接表的复合主键，同时单独充当外键。

未正确建立的多对多关系是**无法解析**的。图 1-7 演示了一个无法解析的多对多关系，其中 Customers 表中的一行可能关联到 Entertainers 表中的多行，同时 Entertainers 表中的一行可能关联到 Customers 表中的多行。

Customers

CustomerID	CustFirstName	CustLastName	CustPhone	<< 其他列 >>
10001	Doris	Hartwig	555-2671	…
10002	Deb	Waldal	555-2496	…
10003	Peter	Brehm	555-2501	…

Entertainers

EntertainerID	EntertainerName	EntertainerPhone	<< 其他列 >>
1001	Carol Peacock Trio	555-2691	…
1002	Topazz	555-2591	…
1003	JV & the Deep Six	555-2511	…

图 1-7　无法解析的多对多关系

这个关系是无法解析的，这是由于多对多关系中存在的固有问题导致的。具体地说，怎么能轻松地将第一个表中的行关联到第二个表中的行呢？换而言之，如何将单个顾客关联到多个演唱组合或将特定的演唱组合关联到多位顾客呢？（如果你运营一家演出经纪公司，肯定希望顾客都与多个演唱组合签约，而演唱组合都与多位顾客签约！）将 Customers 表中的一些列插入到 Entertainers 表中，还是将 Entertainers 表中的一些列插入到 Customers 表中呢？当你处理相关的数据时，这两种做法都会带来一些问题，更别说数据完整性方面的问题了。这个难题的解决方案是按前面说的那样创建一个链接表。通过创建并使用链接表，可正确地解析多对多关系，如图 1-8 所示。

Customers

CustomerID	CustFirstName	CustLastName	CustPhone	<< 其他列 >>
10001	Doris	Hartwig	555-2671	…
10002	Deb	Waldal	555-2496	…
10003	Peter	Brehm	555-2501	…

Engagements（链接表）

EngagementID	CustomerID	EntertainerID	StartDate	<< 其他列 >>
43	10001	1001	2007-10-21	…
58	10001	1002	2007-12-01	…
62	10003	1005	2007-12-09	…
71	10002	1003	2007-12-22	…
125	10001	1003	2008-02-23	…

Entertainers

EntertainerID	EntertainerName	EntertainerPhone	<< 其他列 >>
1001	Carol Peacock Trio	555-2691	…
1002	Topazz	555-2591	…
1003	JV & the Deep Six	555-2511	…

图 1-8　正确地解析多对多关系

在图 1-8 中，链接表是以 Customers 表中的 CustomerID 和 Entertainers 表中的 EntertainerID 为基础创建的。与数据库中的其他表一样，这个新的链接表也有名称——Engagements。实际上，Engagements 是一个典型的存储事件信息的表。演唱组合 1003（JV & the Deep Six）在 2 月 23 日为顾客 10001（Doris Hartwig）演出了。链接表让你能够存储有关这种关联的额外信息，如日期，还可能有演出合约的价格。链接表的真正优势在于，让你能够将多对多关系涉及的两个表中的任意数量的行关联起来。正如这个示例演示的，你现在可轻松地将给定顾客关联到任意数量的演唱组合，还可将特定的演唱组合关联到任意数量的顾客。

前面说过，搞明白关系对使用多表 SQL 查询大有裨益，因此开始阅读本书第三部分前，务必回过头来再次阅读本节。

1.4 学习关系型数据库有何好处

为何要搞明白关系型数据库呢？为何要关心数据处理环境呢？另外，学习关系型数据库有何好处呢？下面来对此做出颇有启迪而有趣的回答。

花时间学习关系型数据库犹如投资，将让你获得独特的优势。你必须对关系型数据库有深入的认识，因为这是当前使用最广泛的数据模型。将行业杂志和信息技术服务部门的说法忘在脑后吧，当前企业和组织使用的大量数据依然是通过关系型数据库收集、维护和操作的。诚然，关系模型扩展层出不穷，处理关系型数据库的程序引入了面向对象功能，关系型数据库也被集成到 Web 和云中，但不管你如何处理，关系型数据库依然是关系型数据库！关系型数据库有 40 多年的历史，但依然处于日益强大的阶段，在可预见的将来不可能被取代。

当今使用的商业数据库管理软件几乎都是关系型的，尽管 C. J. Date 和 Fabian Pascal 等人可能质疑是否有纯粹的关系型数据库实现。要在数据库领域找到一份报酬丰厚的工作，必须知道如何设计关系型数据库以及如何使用流行的 RDBMS 程序来实现它。同时，鉴于当前依赖于互联网、云服务和互联服务的公司和企业如此之多，你还必须具备 Web 开发经验。

从很多方面说，对关系型数据库有深入的认识都大有裨益。例如，对关系型数据库设计越熟悉，开发特定数据库的应用程序就越容易。知道 RDBMS 程序为何提供配套的工具以及如何最大限度地利用这些工具后，你将发现 RDBMS 程序是多么地直观。熟悉关系型数据库将为你学习如何使用 SQL 提供极大的帮助，因为 SQL 是创建、维护和操作关系型数据库的标准语言。

接下来该怎么办

知道学习关系型数据库的重要性后，你必须明白**数据库理论**和**数据库设计**之间是有差别的。数据库理论涵盖相关的原理和规则，是关系型数据库模型的基石；但在现实世界的“黑暗洞穴”中，你在神圣的学术殿堂中学到的这些知识将马上黯然失色。虽然如此，理论依然很重要，因为它保证了关系型数据库的结构是合理的，而且对数据库中数据执行的所有操作的结果都是可预测的。数据库设计指的是一系列组织有序的关系型数据库设计步骤。良好的数据库设计方法有助于确保数据库数据的完整性、一致性和准确性，还可保证你获取的任何信息都尽可能是准确的、最新的。

如果你要设计并创建企业级数据库、开发基于 Web 的互联网商业数据库或深入学习数据仓库技术，就应考虑深入地学习数据库理论。但即便你不打算探索这些领域，而想成为高端数据库顾问，也应考虑这样做。如果你的目标只是在各种平台上设计并创建关系型数据库（我想本书的读者多半如此），只需学习良好而可靠的数据库设计方法就够了。千万别忘了，设计数据库相对容易，但在特定平台上使用特定 RDMBS 程序**实现**数据库则是另外一码事。

市面上介绍数据库设计的图书很多，其中有些只涉及了数据库设计方法，如 Mike Hernandez 编著的 *Database Design for Mere Mortals* 第 3 版只讨论了数据库设计方法，而有些兼顾了数据库理论和数据库设计，如 C. J. Date 编著的 *An Introduction to Database Systems* 第 8 版。需要指出的是，讨论数据库理论的图书阅读起来并不一定轻松。确定研究方向后，选择并购买合适的图书，冲上一杯咖啡或倒上一杯其他的饮料，全神贯注地去阅读。熟悉关系型数据库后，你就需要学习并精通 SQL。

这正是你阅读本书的原因。

1.5　小结

本章首先简要地讨论了当今常见的各种数据库。你了解到处理动态数据的组织使用操作型数据库，以确保获取的信息尽可能是准确且最新的，而处理静态数据的组织使用的则是分析型数据库。

然后简要地介绍了关系型数据库模型的历史。你了解到这种模型是 E. F. Codd 博士基于一些数学分支推出的，到现在已有将近 50 年的历史。你知道，自 20 世纪 70 年代起，人们开发了用于各种计算环境的数据库软件，这些软件的威力、性能和功能在不断增强。从大型机到台式机，从 Web 到联网服务，RDBMS 程序已成为当今众多组织的支柱。

接下来对关系型数据库进行了剖析。你了解了关系型数据库的基本元素以及这些元素的用途，还学习了三种关系，知道它们对理解数据库结构和 SQL 来说都很重要。

最后阐述了熟悉关系型数据库及其设计有何好处。你了解到关系型数据库是当今用得最多的数据库类型，你遇到的几乎每款数据库程序支持的都是关系型数据库。你还对如何更深入地了解数据库理论和设计有了大致的认识。

下一章将介绍一些微调数据库结构的技巧。

第 2 章 确保数据库结构合理

“我们塑造建筑，而建筑又反过来塑造我们。”

——温斯顿·丘吉尔爵士

本章涵盖如下主题：

- 为何在本书开头讨论数据库设计
- 为何要关心数据库结构是否合理
- 微调列
- 微调表
- 建立合理的关系
- 就这些吗
- 小结

本书的大多数读者很可能使用的是既有数据库，且它们是使用你喜欢的 RDBMS 程序实现的（但愿如此）。当前，我无法判断你（或者说数据库开发人员）是否具备妥善地设计数据库所需的知识、技能和时间，因此只能做最坏的打算：你面临的很多表都有微调空间。所幸你即将学习一些技巧，这些技巧能够确保数据库结构合理，从而让你能够轻松地从表中获取所需的信息。

2.1 为何在本书开头讨论数据库设计

你可能会问：为何在本书开头讨论数据库设计呢？原因很简单：如果数据库结构设计不善，本书后面介绍的很多 SQL 语句都将难以实现，甚至毫无用处。然而，如果数据库结构设计良好，本书介绍的很多技能都将给你提供极大的帮助。

本章不介绍数据库设计的难点，而旨在帮助你确保数据库结构合理。强烈建议你详细阅读本章，这样你就能够确保表的结构正确而合理了。

> ❖ **说明** 这里只讨论如何确保数据库设计合乎逻辑，明白这一点很重要。这里不介绍如何在支持 SQL 的数据库管理系统中创建或实现数据库，因为正如本书前言中指出的，这些主题不在本书的讨论范围之内。

2.2 为何要关心数据库结构是否合理

如果数据库结构不合理，即便从数据库中检索看似简单的信息也会遇到麻烦，而且数据处理起来会很困难，同时每当需要增删列时你都会畏手畏脚。结构设计不合理还会影响数据库的其他方面，如数据完整性、表间关系以及检索精确信息的能力。这些只是冰山一角，问题还有很多！为避免诸如此类的麻烦，务必确保数据库结构合理。

如果一开始就妥善地设计数据库，很多问题都可避免。即便数据库已经设计好了，你依然可以采取下面介绍的方法来获得结构合理带来的好处。然而，你必须明白，数据库结构的最终质量取决于你花了多少时间来微调。你越有耐

心，越细心，获得成功的可能性就越大。

下面来介绍调整数据库结构的第一步：微调列。

2.3 微调列

鉴于列是最基本的数据库结构，因此将表作为一个整体进行微调前，必须确保列处于最佳状态。通过微调列，通常可消除表存在的一些问题，同时避免其他可能出现的潜在问题。

2.3.1 列名的组成

前一章说过，列是其所属表表示的客体的一个特征。通过给列指定合适的名称，就能指出它表示的特征。模糊不清的列名肯定会带来麻烦，同时意味着没有仔细考虑列的用途。请按如下清单检查每个列名。

- ❑ **列名是描述性的且整个组织都明白其含义？**如果数据库要供多个部门的用户使用，务必选择每个用户都明白的列名。语义是个很有意思的东西，使用对不同的人来说含义不同的单词无疑是自找麻烦。
- ❑ **列名清晰而明确？**PhoneNumber 是个极具误导性的列名。这个列表示的是哪种电话号码呢？家庭电话、工作电话还是手机？请学会做到具体而明确。如果需要记录所有这些类型的电话号码，就创建 HomePhone、WorkPhone 和 CellPhone 列。

> ❖ **说明** 有一种观点认为，HomePhone、WorkPhone 和 CellPhone 实际上是一个重复组（repeating group），应将它们移到一个独立的表中。该表可存储相关客户或员工的多个电话号码，还可包含表明电话类型的列，让相关的人员或公司可以有任意数量的电话号码。有关这一点，将在下一节讨论表结构时更详细地介绍。

除确保列名清晰而明确外，还应避免在多个表中使用相同的列名。假设有 3 个表，分别为 Customers、Vendors 和 Employees。这些表肯定都包含表示城市（City）和州（State）的列，如果在这些表中，这些列的名称相同，那么等你需要引用特定的列时，将遇到麻烦。例如，如何区分这 3 个表中的 City 列呢？答案很简单：在每个列名中添加简短的前缀。例如，在 Vendors 表中使用列名 VendCity，在 Customers 表中使用列名 CustCity，在 Employees 表中使用列名 EmpCity。这样就能轻松而明确地引用特定的列了（这种技巧适用于任何通用列，如 FirstName、LastName 和 Address）。

这里需要牢记的要点如下：确保数据库中每个列的名称都是独一无二的，且在整个数据库结构中只出现一次。对于这条规则，唯一的例外是用来在两个表之间建立关系的列。

> ❖ **说明** 在表中使用前缀的规模因人而异。表包含通用的列名时，有些数据库设计人员只给通用名称添加前缀，而有些人会给这个表中的所有列名都添加前缀。不管选择使用哪种方法，务必在整个数据库结构中保持一致，这至关重要。

- ❑ **列名中使用了首字母缩写或缩略语吗？**如果使用了，请务必修改！首字母缩写难以理解，并且很容易误导人。看到列名 CAD_SW 时，你知道它表示的是什么吗？请慎用缩略语，就算要使用也请小心处理。仅当缩略语能让列名的含义更明确时才使用它，千万不要使用让列名的含义变得模糊的缩略语。
- ❑ **列名明确或含蓄地指出列表示多个特征吗？**这样的列名很容易识别，因为它们通常包含单词 and 或 or。另外，千万不要使用包含反斜杠（\）、连字符（-）或和号（&）的列名。如果发现列名为 Phone\Fax 或 Area or Location，请务必检查列中存储的数据，进而确定是否要将其分解为多个更小的列。

> ❖ **说明** SQL 标准定义了**常规标识符**（regular identifier）和**分隔式标识符**（delimited identifier）。常规标识符指的是这样的名称，即必须以字母打头，且只能包含字母、数字和下划线（不能包含空格）；分隔式标识符指的是用双引号括起的名称，必须以字母打头，且只能包含字母、数字、下划线、空格以及其他几种特殊字符。给列命名时，建议你只使用常规标识符命名规则，因为很多 SQL 实现只支持常规标识符命名规则。

根据上述检查清单修订列名后，就只余下一项任务了：确保使用单数型列名。复数型列名（如 Categories）意味着在给定行中相应的列可能包含多个值，这可不是什么好主意。列名为何必须是单数型的呢？因为一个列表示它所属表表示的客体的一个特征。相反，表名必须是复数型的，因为一个表包含一系列类似的物件或事件。使用这种命名约定后，可轻松地区分表名和列名。

❖ **说明** 虽然我推荐使用 SQL 标准指定的命名约定，但别忘了，当你（或负责实现数据库的数据库开发人员）在特定的 RDBMS 应用程序中实现数据库时，列名可能会发生变化，因为列名必须遵循该 RDBMS 的命名约定。

2.3.2 消除不完善的地方

整理好列名后，将重点转向列本身的结构。你可能非常确定列是合理的，但依然可以采取一些措施来确保它们尽可能高效。请按如下清单检查各个列，以判断是否要对其做进一步调整。

- ❑ **确保每个列都是当前表表示的客体的一个特征**。这里的理念是判断该列在当前表中是否是不可或缺的，如果不是，就将其删除或移到另一个表中。对于这一规则，仅有的两个例外是用于在当前表和数据库中的其他表之间建立关系的列，还有用于让数据库应用程序能够执行特定任务的列。例如，在图 2-1 所示的 Classes 表中，StaffLastName 和 StaffFirstName 就是多余的，因为这个表包含 StaffID 列。StaffID 列用于在 Classes 和 Staff 表之间建立关系。要同时查看这两个表中的数据，可使用视图或 SQL SELECT 查询。对于表中多余的列，可将其删除，也可以这些列为基础新建一个表（如果它们没有出现在数据库的其他地方），这将在本章后面介绍。

Staff

StaffID	StaffFirstName	StaffLastName	StaffStreetAddress	StaffCity	StaffState	<< 其他列 >>
98014	Peter	Brehm	722 Moss Bay Blvd.	Kirkland	WA	...
98019	Mariya	Sergienko	901 Pine Avenue	Portland	OR	...
98020	Jim	Glynn	13920 S.E. 40th Street	Bellevue	WA	...
98021	Tim	Smith	30301 166th Ave. N.E.	Seattle	WA	...
98022	Carol	Viescas	722 Moss Bay Blvd.	Kirkland	WA	...
98023	Alaina	Hallmark	Route 2, Box 203 B	Woodinville	WA	...

Classes

ClassID	Class	ClassroomID	StaffID	StaffLastName	StaffFirstName	<< 其他列 >>
1031	Art History	1231	98014	Brehm	Peter	...
1030	Art History	1231	98014	Brehm	Peter	...
2213	Biological Principles	1532	98021	Smith	Tim	...
2005	Chemistry	1515	98019	Sergienko	Mariya	...
2001	Chemistry	1519	98023	Hallmark	Alaina	...
1006	Drawing	1627	98020	Glynn	Jim	...
2907	Elementary Algebra	3445	98022	Viescas	Carol	...

图 2-1 一个包含多余列的表

- ❑ **确保每个列值包含单个值**。可能存储多个同类型值的列被称为**多值列**，如包含多个电话号码的列。同理，可能存储多个不同值的列被称为**多部分列**，如包含商品编号和商品描述的列。多值列和多部分列可能严重破坏数据库，在你想要对数据进行编辑、删除或排序时尤其如此。通过确保每列都只存储单个值，可在很大程度

上确保数据的完整性和信息的准确性。但就目前而言，你只需找出所有的多值列和多部分列，并将它们记录下来，至于如何对它们进行分解，将在下一节介绍。

- **确保列存储的不是计算结果或拼接结果**。在设计良好的表中，不可包含计算得到的列。计算列的问题在于其值。不同于电子表格中的单元格，列并不存储计算公式本身，因此当计算公式的任何一部分的值发生变化时，计算列的值并不会相应地更新。要更新计算列的值，要么手动更新，要么编写自动完成这种任务的代码，而这两种方式都要求用户或开发人员承担确保这些值得以更新的职责。要处理计算结果，更佳的方式是使用 SELECT 语句，这样做的优点将在第 5 章介绍。
- **确保每个列在整个数据库中都只出现一次**。如果你犯了常见的错误，将同一列（如 CompanyName）插入到了数据库的多个表中，将面临数据不一致的问题。当你在一个表中修改一个列，而忘记对出现在其他地方的该列做同样的修改时，这种问题就会显现出来。通过确保每个列在整个数据库中都只出现一次，可将这种问题“斩草除根”。仅当使用列在表之间建立关系时，才可以违反这个规则。

> ❖ **说明**　*在有些商用数据库管理系统的最新版本中，可将列定义为一个表达式的计算结果。如果你使用的数据库系统支持计算列，就可定义这样的列，但别忘了，每当表达式包含的列的值发生变化或在你获取包含计算列的行时，数据库系统都将消耗更多的资源来确保计算列的值是最新的。*

2.3.3　分解多部分列

前面说过，多部分列和多值列会严重破坏数据完整性，因此必须对它们进行分解，以消除所有潜在的问题。先分解多部分列还是先分解多值列呢？你可以随便选择，因此这里先介绍如何分解多部分列。

只需回答一些非常简单的问题，就可判断出一个列是否是多部分列：对于这个列当前包含的值，可将其分解成多个不同的部分吗？提取某部分信息时，是否会由于它位于包含其他信息的列中而遇到麻烦？只要其中有一个问题的答案是肯定的，这个列就是多部分列。图 2-2 是一个设计得很糟糕的表，其中包含多个多部分列。

Customers

CustomerID	CustomerName	StreetAddress	PhoneNumber	<< 其他列 >>
1001	Suzanne Viescas	15127 NE 24th, #383, Redmond, WA 98052	425 555-2686	...
1002	William Thompson	122 Spring River Drive, Duvall, WA 98019	425 555-2681	...
1003	Gary Hallmark	Route 2, Box 203B, Auburn, WA 98002	253 555-2676	...
1004	Robert Brown	672 Lamont Ave, Houston, TX 77201	713 555-2491	...
1005	Dean McCrae	4110 Old Redmond Rd., Redmond, WA 98052	425 555-2506	...
1006	John Viescas	15127 NE 24rh, #383, Redmond, WA 98052	425 555-2511	...
1007	Mariya Sergienko	901 Pine Avenue, Portland, OR 97208	503 555-2526	...
1008	Neil Patterson	233 West Valley Hwy, San Diego, CA 92199	619 555-2541	...

多部分列

图 2-2　一个包含多部分列的表

图中所示的 Customers 表包含两个多部分列：CustomerName 和 StreetAddress。另外，还有一个列可能是多部分的：PhoneNumber。如何根据姓或邮政编码对数据进行排序呢？如何根据州查找数据呢？你没法这样做，因为这些值嵌在包含其他信息的列中。如你所见，每个列都可分解为更小的列。例如，CustomerName 可分解为两个列：CustFirstName 和 CustLastName（请注意，这里遵循了本章前面讨论的命名约定，给 FirstName 和 LastName 列添加了前缀 Cust）。找出表中的多部分列后，确定它存储的值由几个部分组成，再将这个列分解为相应数量的列。图 2-3 演示了如何对 Customers 表中的两个多部分列进行分解。

Customers

CustomerID	CustFirstName	CustLastName	CustAddress	CustCity	CustState	CustZipcode
1001	Suzanne	Viescas	15127 NE 24th, #383	Redmond	WA	98052
1002	William	Thompson	122 Spring River Drive	Duvall	WA	98019
1003	Gary	Hallmark	Route 2, Box 203B	Auburn	WA	98002
1004	Robert	Brown	672 Lamont Ave	Houston	TX	77201
1005	Dean	McCrae	4110 Old Redmond Rd.	Redmond	WA	98052
1006	John	Viescas	15127 NE 24th, #383	Redmond	WA	98052
1007	Mariya	Sergienko	901 Pine Avenue	Portland	OR	97208
1008	Neil	Patterson	233 West Valley Hwy	San Diego	CA	92199

图 2-3 分解 Customers 表中的多部分列

❖ **说明** 除了对 CustomerName 和 StreetAddress 进行分解外，对于存储北美电话号码的数据库，将 PhoneNumber 分解为两列（区号和本地电话号码）也是个不错的主意。在其他国家，将电话号码中的城市代码分离出来可能很有用。实际上，大多数商务数据库将电话号码存储在一列中，但对于用于分析人口数据的数据库，将区号或城市代码分离出来可能很重要。遗憾的是，由于篇幅有限，我无法在图 2-3 中演示这一点。

在有些情况下，可能难以识别多部分列。请看图 2-4 所示的 Instruments 表。乍一看，好像根本没有多部分列，但仔细研究后，你将发现 InstrumentID 实际上是一个多部分列。该列存储的值包含两项信息：乐器所属的类别［如 AMP（功放）、GUIT（吉他）和 MFX（多效乐器）］及其标识号。为确保数据完整性，应将这两个值存储在不同的列中。想象一下，如果要更新这个列，将类别 MFX 改为 MFU，将非常困难。为此，你必须编写代码对这个列的值进行分析，如果其中包含 MFX，就将其替换为 MFU。这并没有难到无法完成的程度，但无疑增加了不必要的麻烦；倘若数据库设计良好，这样的麻烦完全可以避免。遇到像这里这样的列时，请将它们分解为更小的列，确保列结构合理而高效。

Instruments

InstrumentID	Manufacturer	InstrumentDescription	<< 其他列 >>
GUIT2201	Fender	Fender Stratocaster	...
MFX3349	Zoom	Player 2100 Multi-Effects	...
AMP1001	Marshall	JCM 2000 Tube Super Lead	...
AMP5590	Crate	VC60 Pro Tube Amp	...
SFX2227	Dunlop	Cry Baby Wah-Wah	...
AMP2766	Fender	Twin Reverb Reissue	...

图 2-4 一个微妙的多部分列

2.3.4 分解多值列

多部分列分解起来并不太难，但多值列分解起来可能更麻烦，需要花些功夫。所幸多值列识别起来很容易：在这种列存储的数据中，几乎都包含逗号、分号或其他常见的分隔符，这些分隔符用于分隔不同的值。图 2-5 展示了一个多值列。

在这个示例中，每位飞行员都可能获得任意数量的不同型号飞机的驾驶证，而这些驾驶证存储在单个名为 Certifications 的列中。这个列存储数据的方式带来了极大的麻烦，因为与多部分列一样，你肯定会遇到同样的数据完整性问题。如果仔细查看数据，你将发现要在 SQL 查询中根据这个列进行查找或排序会非常困难。要以合理的方式分解这个列，必须明白多值列与其最初所属表之间的关系。

Pilots

PilotID	PilotFirstName	PilotLastName	HireDate	Certifications	<< 其他列 >>
25100	Sam	Alborous	1994-07-11	727, 737, 757, MD80	...
25101	Jim	Wilson	1994-05-01	737, 747, 757	...
25102	David	Smith	1994-09-11	757, MD80, DC9	...
25103	Kathryn	Patterson	1994-07-11	727, 737, 747, 757	...
25104	Michael	Hernandez	1994-05-01	737, 757, DC10	...
25105	Kendra	Bonnicksen	1994-09-11	757, MD80, DC9	...

图 2-5　一个包含多值列的表

在多值列的值与其父表的每一行之间，存在多对多关系：多值列中的一个值可能与父表的任意数量的行相关联，而父表中的一行可能与多值列中任意数量的值相关联。例如，在图 2-5 中，Certifications 列中的飞机型号可能与任意数量的飞行员相关联，而一位飞行员可能与 Certifications 列中任意数量的飞机型号相关联。对于这个多对多关系，可像分解数据库中的其他多对多关系一样进行分解，即使用链接表。

要创建链接表，可以多值列和原表中的主键列为基础创建一个新表。给新创建的链接表指定合适的名称，并将这两个列指定为复合主键（在这里，这两列的值一道唯一地标识了新表中的每一行）。这样，就可以一对一的方式将新表中两列的值关联起来了。图 2-6 以图 2-5 所示的 Pilots 表为例演示了如何创建链接表。

Pilots

PilotID	PilotFirstName	PilotLastName	HireDate	<< 其他列 >>
25100	Sam	Alborous	1994-07-11	...
25101	Jim	Wilson	1994-05-01	...
25102	David	Smith	1994-09-11	...
25103	Kathryn	Patterson	1994-07-11	...
25104	Michael	Hernandez	1994-05-01	...
25105	Kendra	Bonnicksen	1994-09-11	...

Pilot_Certifications（链接表）

PilotID	CertificationID
25100	8102
25100	8103
25100	8105
25100	8106
25101	8103
25101	8104
25101	8105

Certifications

CertificationID	TypeofAircraft	<< 其他列 >>
8102	Boeing 727	...
8103	Boeing 737	...
8104	Boeing 747	...
8105	Boeing 757	...
8106	McDonnell Douglas MD80	...

图 2-6　通过使用链接表来分解多值列

请对照旧表 Pilots 和新表 Pilot_Certifications 中与 Sam Alborous（PilotID 25100）相关的条目。新的链接表的主要优点在于，现在可将一位飞行员与任意数量的驾驶证相关联；另外，有些问题回答起来要容易得多。例如，可确定哪

些飞行员有驾驶波音 747 飞机的资质，或者检索特定飞行员拥有的驾驶证列表。另外，还可以以任何顺序对数据进行排序，而不会有任何负面影响。

❖ **说明**　在有些数据库管理系统（其中最著名的是 Microsoft Office Access 2007 及更高版本）中，可以显式地定义多值列。然而，为了支持这一点，这种数据库系统将创建一个隐藏的系统表，它类似于图 2-6 所示的链接表。坦率地说，我希望看到并控制表的设计，因此建议你自己动手创建正确的数据结构，而不要依赖数据库提供的相关功能。

只要遵照本节介绍的流程完成相关的工作，就可确保列的结构合理。调整好列后，下面转向第二项任务，看看表的结构。

2.4　微调表

你创建的所有 SQL 查询都是以表为基础的。创建多表 SQL 查询时，你很快就会发现，设计糟糕的表会导致数据完整性问题且难以处理，因此必须以尽可能高效的方式构建表，这样才能轻松地检索所需的信息。

2.4.1　表名的组成

2.3 节介绍了对列来说合适的名称有多重要，还介绍了给列命名时为何要三思而后行。你很快就会知道，对表来说亦如此。根据定义，表应该表示单种客体，如果它表示的是多种客体，就应将其分成多个小表。表名必须清楚地指出它表示的客体。如果表名不明确、模糊或不清晰，就可以肯定设计者未深入考虑表所表示的客体。请按如下清单检查表名，确保它们是合理的。

- ❑ **表名是否是唯一的，且描述性足够高，组织的任何成员都明白其含义？**通过给表指定独一无二的名称，可确保数据库中的每个表都表示不同的客体，且组织的每个成员都明白它表示的是什么。为定义独一无二的描述性名称，确实需要花些功夫，但长远看这种付出是非常值得的。
- ❑ **表名是否准确、清楚而明确地指出了表表示的客体？**如果表名模糊不清，就可以肯定表表示的是多个客体。例如，Dates 就是一个模糊的表名，仅通过它难以准确地判断表表示的是什么，除非你手里有有关表的描述。我们假设这个表出现在娱乐公司使用的数据库中。如果你仔细查看这个表，可能发现其中包含客户会面日期和艺人预约演出日期。显然，这个表表示的是两个客体。为消除这个问题，可将这个表分成两个表，并给它们指定合适的名称，如 Client_Meetings 和 Entertainer_Schedules。
- ❑ **表名是否包含表示物理特征的单词？**在表名中，请避免使用诸如 File、Record 和 Table 这样的单词，因为它们会造成迷惑。包含这类单词的表名很可能表示的是多个客体。请看表名 Employee_Record。从表面上看，这个表名没有任何问题，但如果你想想员工记录该表示什么，就会意识到存在一些潜在的问题。这个表名包含我们要极力避免的单词，它可能表示三个客体：员工、部门和薪酬。考虑到这一点，应将原表（Employee_Record）分成三个表，每个客体一个。
- ❑ **是否将缩略语或缩写用作了表名？**如果答案是肯定的，赶快修改！缩写很少能表达出表所表示的客体，而缩略语通常难以理解。假设贵公司的数据库中有一个名为 SC 的表，在不知道这两个字母本身的含义的情况下，怎么知道这个表表示的是什么呢？实际上，你无法轻松地确定这个表表示的客体。另外，你可能发现，对公司的不同部门来说，这个表可能意味着不同的东西（这太可怕了）：人事部门的人认为它表示的是 Steering_Committees，信息系统部门的人认为它表示的是 System_Configurations，而安全部门的人认为它指的是 Security_Codes。在表名中应避免使用缩写和缩略语，这个示例明确地证明了这一点。
- ❑ **表名显式或隐式地表示多种客体吗？**这是给表命名时最常犯的错误之一，且比较容易识别。这种表名通常包含单词 and 或 or，以及诸如反斜杠（\）、连字符（-）或和字符（&）等字符，如 Facility\Building 和 Department or Branch。如果表是以这种方式命名的，必须确定它是否表示的是多种客体；如果答案是肯定的，就将其分

成多个小表，并给每个小表指定合适的名称。

❖ **说明**　别忘了，SQL 标准定义了**常规标识符**，它指的是必须以字母打头且只能包含字符、数字和下划线的名称。这种标识符不能包含空格。SQL 标准还定义了**分隔式标识符**，它指的是用双引号括起的名称，必须以字母打头，且只能包含字母、数字、下划线、空格以及其他几种特殊字符。给表命名时，建议你只使用常规标识符命名规则，因为很多 SQL 实现只支持常规标识符命名规则。

完成表名修订工作后，还有一项任务需要完成：再次检查表名，确保使用的是复数形式。为何要使用复数形式呢？因为表存储的是它表示的客体的**一系列实例**。例如，Employees 表存储的是众多员工（而不是单个员工）的数据。使用复数形式还可帮助你将表名和列名区分开来。

❖ **说明**　对数据库进行逻辑设计时，使用复数形式的表名是一条非常不错的指导原则。这让你能够非常轻松地将表名和列名区分开来，将它们显示到投影屏幕或写在会议室的白板上时尤其如此。

然而，别忘了，当你（或负责实现数据库的数据库开发人员）着手在特定的 RDBMS 应用程序中实现数据库时，可能需要修改表名。届时，表名必须遵循开发人员在该 RDBMS 中都遵循的命名约定。

2.4.2　确保结构合理

修订表名后，下面将重点转向表结构。必须妥善地设计表，以便能够高效地存储数据以及检索到准确的信息。需要创建复杂的多表 SQL 查询时，多花些时间确保表的设计良好将带来丰厚的回报。请根据如下清单来判断表结构是否合理。

- **确保表表示的是单种客体**。我知道，这一点我已经说过很多次了，但怎么强调都不过分。只要确保每个表表示的都是单种客体，就可极大地降低出现数据完整性问题的潜在风险。另外，别忘了表表示的客体可以是物件，也可以是事件。所谓物件，指的是看得见摸得着的东西，如员工、供应商、机器、建筑物或部门。而事件指的是在特定时点发生的事情，具有需要记录的特征。每个人都可能遇到的典型事件是看医生。虽然看医生这一事件摸不着，但它确实具备需要记录的特征，如看病日期、患者血压和患者体温。
- **确保每个表都有一个主键**。对于每个表，都必须给它指定一个主键，其原因有两个。首先，主键唯一地标识了表中的每一行；其次，主键用于在表之间建立关系。如果不给每个表都指定一个主键，终将遇到数据完整性问题，还将在创建某些类型的多表 SQL 查询时遇到麻烦。本章后面将提供一些有关如何定义合适主键的小贴士。
- **确保表中没有多部分列和多值列**。从理论上说，这些问题在你调整列结构时就已解决了。尽管如此，再次检查各列，确保消除了所有的多部分列和多值列仍不失为一个好主意。
- **确保表中没有计算列**。虽然你可能深信表结构中现在根本没有计算列，但在调整列的过程中，你可能忽略了一两个。此时再次查看表结构，并将可能遗漏的所有计算列都消除正当其时。
- **确保表中没有多余的重复列**。表设计糟糕的迹象之一是包含来自其他表的重复列。基于以下两个原因，你可能觉得必须在表中添加重复的列：提供参考信息；指出某个特定类型的值出现了多次。别忘了，前面谈到了 HomePhone、WorkPhone 和 CellPhone 可能是重复的列。当你处理数据并尝试从表中检索数据时，这些重复列会带来各种麻烦。下面来看看如何处理重复列。

2.4.3　消除多余的重复列

就确保结构良好而言，最难的部分可能是处理重复列。下面通过两个示例来演示消除重复列的正确方式。

在图 2-7 所示的表中，包含提供参考信息的重复列。

Staff

StaffID	StaffFirstName	StaffLastName	StaffStreetAddress	StaffCity	StaffState	<< 其他列 >>
98014	Peter	Brehm	722 Moss Bay Blvd.	Kirkland	WA	...
98019	Mariya	Sergienko	901 Pine Avenue	Portland	OR	...
98020	Jim	Glynn	13920 S.E. 40th Street	Bellevue	WA	...
98021	Tim	Smith	30301 166th Ave. N.E.	Seattle	WA	...
98022	Carol	Viescas	722 Moss Bay Blvd.	Kirkland	WA	...
98023	Alaina	Hallmark	Route 2, Box 203 B	Woodinville	WA	...

这些列是多余的

Classes

ClassID	Class	ClassroomID	StaffID	StaffLastName	StaffFirstName	<< 其他列 >>
1031	Art History	1231	98014	Brehm	Peter	...
1030	Art History	1231	98014	Brehm	Peter	...
2213	Biological Principles	1532	98021	Smith	Tim	...
2005	Chemistry	1515	98019	Sergienko	Mariya	...
2001	Chemistry	1519	98023	Hallmark	Alaina	...
1006	Drawing	1627	98020	Glynn	Jim	...
2907	Elementary Algebra	3445	98022	Viescas	Carol	...

图 2-7　一个包含重复列的表，添加这些重复列旨在提供参考信息

在这个示例中，Classes 表包含 StaffLastName 和 StaffFirstName 列，旨在让查看这个表的人能够看到课程教员的姓名。这些列是多余的，因为 Classes 和 Staff 表之间存在一对多关系（一位教员可讲授任意数量的课程，但一门课程只由一名教员讲授）。这个关系是由 StaffID 建立的，它让你能够在 SQL 查询中同时查看这两个表中的数据。考虑到这一点，完全可以将 StaffLastName 和 StaffFirstName 列从 Classes 表中删除，而不会有任何负面影响。图 2-8 显示了修订后的 Classes 表结构。

Staff

StaffID	StaffFirstName	StaffLastName	StaffStreetAddress	StaffCity	StaffState	<< 其他列 >>
98014	Peter	Brehm	722 Moss Bay Blvd.	Kirkland	WA	...
98019	Mariya	Sergienko	901 Pine Avenue	Portland	OR	...
98020	Jim	Glynn	13920 S.E. 40th Street	Bellevue	WA	...
98021	Tim	Smith	30301- 166th Ave. N.E.	Seattle	WA	...
98022	Carol	Viescas	722 Moss Bay Blvd.	Kirkland	WA	...
98023	Alaina	Hallmark	Route 2, Box 203 B	Woodinville	WA	...

Classes

ClassID	Class	ClassroomID	StaffID	<< 其他列 >>
1031	Art History	1231	98014	...
1030	Art History	1231	98014	...
2213	Biological Principles	1532	98021	...
2005	Chemistry	1515	98019	...
2001	Chemistry	1519	98023	...
1006	Drawing	1627	98020	...
2907	Elementary Algebra	3445	98022	...

图 2-8　消除重复的参考列

如果在这个表中保留这些多余的列，将导致严重的数据不一致问题：必须确保 Classes 表中 StaffLastName 和 StaffFirstName 列的值始终与 Staff 表中相应列的值一致。例如，假设有位女教员结婚了，并决定从此使用丈夫的姓氏。在这种情况下，不仅需要在 Staff 表中修改与该教员对应的行，还必须在 Classes 表中修改她的姓名。这种任务是能够完成的（至少从技术上说如此），但这增加了不必要的负担。另外，使用关系型数据库的一个重要假设是，对于每项数据，都只需在数据库中输入一次即可（唯一的例外情况是使用列在两个表之间建立关系）。与以前一样，最佳措施是将重复列从数据库表中删除。

图 2-9 是另一个包含重复列的表。这个示例错误地使用了重复列来指出特定类型的值出现了多次：使用三个 Committee 列来指出员工是哪些委员会的成员。

Employees

EmployeeID	EmpLastName	EmpFirstName	Committee1	Committee2	Committee3	<< 其他列 >>
7004	Gehring	Darren	Steering			...
7005	Kennedy	John	ISO 9000	Safety		...
7006	Thompson	Sarah	Safety	ISO 9000	Steering	...
7007	Wilson	Jim				...
7008	Seidel	Manuela	ISO 9000			...
7009	Smith	David	Steering	Safety	ISO 9000	...
7010	Patterson	Neil				...
7011	Viescas	Michael	ISO 9000	Steering	Safety	...

图 2-9　一个包含重复列的表，这些重复列用于指出特定类型的值出现了多次

这些重复列会带来问题，其中的原因理解起来比较容易。一个问题是这个表该包含多少个 Committee 列。如果有几位员工是 4 个委员会的成员呢？从这种意义上说，你无法准确地知道将需要多少个 Committee 列。如果有几名员工参加的委员会超过 3 个，就需要在这个表中添加更多的 Committee 列。

第二个问题与从这个表中检索信息相关。如何检索当前为 ISO 9000 委员会成员的员工呢？检索这种信息不是不可能，但有点麻烦。为了准确地回答这个问题，必须执行 3 个不同的查询（或者编写一个检查这三列的查找条件），因为无法确定 ISO 9000 存储在哪个 Committee 列中。这要求你花费更多的精力和时间，而这原本是不必要的。

第三个问题与数据排序相关。无法按委员会对数据进行排序，因为没法按字母顺序排列委员会名称。虽然这些问题看起来好像没什么大不了，但要以特定的顺序查看数据时，它们可能会让你非常懊恼。

如果仔细研究图 2-9 所示的 Employees 表，你很快就会发现员工和委员会之间存在多对多关系：一名员工可以是任意数量委员会的成员，而一个委员会可以由任意数量的员工组成。因此，可以像处理其他多对多关系那样处理这些重复列——创建一个链接表。就 Employees 表而言，可使用主键的副本（EmployeeID）和单个 Committee 列来创建链接表。给这个新表指定合适的名称，如 Committee_Members。将 EmployeeID 和 Committee 列指定为复合主键，再将 Employees 表中的所有 Committee 列都删除，这样就大功告成了（本章后面将更详细地介绍主键）。图 2-10 显示了修订后的 Employees 表及新建的 Committee_Members 表。

Employees

EmployeeID	EmpLastName	EmpFirstName	EmpCity	<< 其他列 >>
7004	Gehring	Darren	Chico	...
7005	Kennedy	John	Portland	...
7006	Thompson	Sarah	Lubbock	...
7007	Wilson	Jim	Salem	...
7008	Seidel	Manuela	Medford	...
7009	Smith	David	Fremont	...
7010	Patterson	Neil	San Diego	...
7011	Viescas	Michael	Redmond	...

Committee_Members

EmployeeID	Committee
7004	Steering
7005	ISO 9000
7005	Safety
7006	Safety
7006	ISO 9000
7006	Steering
7008	ISO 9000
7009	Steering

图 2-10　修订后的 Employees 表及新建的 Committee_Members 表

你消除了 Employees 表中的重复列，但事情还没完。别忘了，员工与委员会之间存在多对多关系。你很可能会问：Committees 表在哪里呢？现在还没有！委员会可能还有其他一些需要记录的特征，如委员会开会的房间以及开会日期。因此，创建一个 Committees 表，并在其中包含诸如 CommitteeID、CommitteeName、MeetingRoom 和 MeetingDay 等列是个不错的主意。创建 Committees 表后，再将 Committee_Members 表中的 Committee 列替换为 Committees 表中的 CommitteeID 列。最终的结构如图 2-11 所示。

Employees

EmployeeID	EmpLastName	EmpFirstName	EmpCity	<< 其他列 >>
7004	Gehring	Darren	Chico	...
7005	Kennedy	John	Portland	...
7006	Thompson	Sarah	Lubbock	...
7007	Wilson	Jim	Salem	...
7008	Seidel	Manuela	Medford	...
7009	Smith	David	Fremont	...
7010	Patterson	Neil	San Diego	...
7011	Viescas	Michael	Redmond	...

Committee_Members

EmployeeID	CommitteeID
7004	103
7005	104
7005	102
7006	102
7006	104
7006	103
7008	104
7009	103

Committees

CommitteeID	CommitteeName	MeetingRoom	MeetingDay
100	Budget	11-C	Tuesday
101	Christmas	9-F	Monday
102	Safety	12-B	Monday
103	Steering	12-D	Tuesday
104	ISO 9000	Main-South	Wednesday

图 2-11　最终的 Employees 表、Committee_Members 表和 Committees 表

通过创建这里所示的表结构，获得了一个真正的好处，那就是可以将一位员工关联到任意数量的委员会，还可以将一个委员会关联到任意数量的员工。这样，就可使用单个 SQL 查询同时查看这 3 个表中的信息。

下面来重温一下前面提到的一个问题：多个名称不同的列可能是一组重复列。请看图 2-12 所示的表。

Employees

EmployeeID	EmpLastName	EmpFirstName	EmpCity	EmpHomePhone	EmpWorkPhone	EmpCellPhone	<< 其他列 >>
7004	Gehring	Darren	Chico	555-1234	556-1234	889-1234	...
7005	Kennedy	John	Portland	555-2345	556-2345	889-2345	...
7006	Thompson	Sarah	Lubbock	555-3456	556-3456	889-3456	...
7007	Wilson	Jim	Salem	555-4567	556-4567		...
7008	Seidel	Manuela	Medford		556-5678	889-5678	...
7009	Smith	David	Fremont	555-5689	556-5689	889-6789	...
7010	Patterson	Neil	San Diego	555-7890	556-7890	889-7890	...
7011	Viescas	Michael	Redmond	555-4321	555-4321		...

图 2-12　Employees 表中重复的电话号码列

你发现了什么潜在的问题吗？首先，在员工没有某种电话号码时，将浪费空间。但还有一个更严重的问题，你能猜到吗？如果员工有两个家庭电话或有传真，该怎么办呢？重要员工不仅自己有手机，公司还给他们配了手机，对于这些员工，该怎么办呢？解决方案是创建一个独立的 Phone_Numbers 表，并将其关联到 Employees 表，如图 2-13 所示。

Employees

EmployeeID	EmpLastName	EmpFirstName	EmpCity	<< 其他列 >>
7004	Gehring	Darren	Chico	...
7005	Kennedy	John	Portland	...
7006	Thompson	Sarah	Lubbock	...
7007	Wilson	Jim	Salem	...
7008	Seidel	Manuela	Medford	...
7009	Smith	David	Fremont	...
7010	Patterson	Neil	San Diego	...
7011	Viescas	Michael	Redmond	...

Phone_Numbers

EmployeeID	PhoneID	PhoneType	PhoneNumber
7004	1	Home	555-1234
7005	2	Home	555-2345
7006	3	Home	555-3456
7007	4	Home	555-4567
7009	5	Home	555-5678
7010	6	Home	555-7890
7011	7	Home	555-4321
7004	8	Work	555-1234

图 2-13　使用独立的相关表解决电话号码问题

采用这种新设计后，就可为每位员工存储任意数量的电话号码了。如果需要存储新类型的电话号码，只需为 PhoneType 列定义一个数据值即可。另外请注意，对于员工 7008，没有浪费存储家庭电话的空间——对于这位员工，根本就没有存储家庭电话的行。还有，注意，在 Phone_Numbers 表中包含一个 PhoneID 列，而在每行中，该列的值都是独一无二的。有关唯一地标识各行的重要性，将在下一节中详细介绍。

微调表结构的过程即将接近尾声，余下的最后一项任务是确保能够唯一地标识各行，且能够唯一地标识数据库中的各个表。

2.4.4 标识是关键

第 1 章说过，主键是最重要的键之一，因为它唯一地标识表中各行，还在整个数据库中标识当前表。它还在两个表之间建立关系。主键的重要性怎么强调都不过分——数据库中的每个表都必须有一个主键！

根据定义，主键是唯一地标识表中各行的一列或多列。主键由单列组成时称为**简单主键**（或简称为主键），而由多列组成时称为**复合主键**。应尽可能定义简单主键，因为建立表关系时，简单主键的效率更高，且使用起来容易得多。仅当必要时才使用复合主键，如定义和创建链接表时。

可将既有的一列或多列用作主键，只要它们满足下面列出的所有标准。当你原本打算要用作主键的一列或多列无法满足所有这些标准时，可将其他列用作主键，或定义一个用作主键的新列。请花点时间根据下面的清单判断数据库中的每个主键是否都是合理的。

- **这些列能够唯一地标识表中各行吗**？表中每行都是表所标识的客体的一个实例。优良的主键确保你能够在其他表中精确地引用当前表中的各行，还有助于避免当前表中包含重复行。
- **这些列包含的值是独一无二的吗**？只要主键的值是独一无二的，就能确保表中没有重复行。
- **这些列是否可能包含未知值**？这一点很重要，因为主键不能包含未知值。只要一列有可能包含未知值，就应撤销其用作主键的资格。
- **这些列的值是否是可选的**？对于值可选的列，不能将其用作主键。列的值可选意味着其值可能是未知的，而前一条指出过，主键不能包含未知值。
- **是否是多部分列**？虽然此时你应该消除了所有的多部分列，但最好再次回答这个问题。如果之前遗漏了多部分列，现在就对其进行处理，并尝试将其他列用作主键，或将分解得到的新列一起用作复合主键。
- **这些列的值是否可能被修改**？主键列的值应保持不变。除非有充分的理由，否则不应修改主键列的值。如果一个列的值可能发生变化，它就很难满足前述各项标准。

前面说过，仅当一列或多列满足上述所有标准时，才能将它们用作主键。在图 2-14 中，PilotID 被用作 Pilots 表的主键，但问题是 PilotID 满足上述所有标准吗？如果满足，这个主键就是合理的，否则要么修改它使其满足前述所有标准，要么将其他列用作主键。

Pilots

PilotID	PilotFirstName	PilotLastName	HireDate	Position	PilotAreaCode	PilotPhone
25100	Sam	Alborous	1994-07-11	Captain	206	555-3982
25101	Jim	Wilson	1994-05-01	Captain	206	555-6657
25102	David	Smith	1994-09-11	FirstOfficer	915	555-1992
25103	Kathryn	Patterson	1994-07-11	Navigator	972	555-8832
25104	Michael	Hernandez	1994-05-01	Navigator	360	555-9901
25105	Kendra	Bonnicksen	1994-09-11	Captain	206	555-1106

图 2-14 将 PilotID 作为主键合理吗

实际上，PilotID 是一个合理的主键，因为它满足前述所有要求。但如果没有能够充当主键的列，该怎么办呢？就拿图 2-15 所示的 Employees 表来说吧，这个表包含可用作主键的列吗？

Employees

EmpLastName	EmpFirstName	EmpCity	EmpState	EmpZip	EmpAreaCode	EmpPhone	HireDate
Gehring	Darren	Chico	CA	95926			1998-12-31
Kennedy	John	Portland	OR	97208	503	555-2621	1998-05-01
Thompson	Sarah	Redmond	WA	98052	425	555-2626	1998-09-11
Wilson	Jim	Salem	OR				1998-12-27
Seidel	Manuela	Medford	OR	97501	541	555-2641	1998-05-01
Smith	David	Fremont	CA	94538	510	555-2646	1998-09-11
Patterson	Neil	San Diego	CA	92199	619	555-2541	1998-05-01
Viescas	Michael	Redmond	WA	98052	425	555-2511	1998-09-11
Viescas	David	Portland	OR	97207	503	555-2633	1998-10-15

图 2-15　这个表有主键吗

这个表没有可用作主键的列（一列或多列），这一点显而易见。除 EmpPhone 外，其他各列都包含重复的值，而 EmpZip、EmpAreaCode 和 EmpPhone 都包含未知值。你可能很想将 EmpLastName 和 EmpFirstName 用作主键，但无法保证新来的员工不会也叫 Jim Wilson 或 David Smith。显然，这个表没有可用作主键的列，因为其中每列的值都可能发生变化。

怎么办呢？你可能想使用每个员工的国家身份号码，如美国的社会保障号或加拿大的社会保险号，但别忘了，虽然罕见，但多人的号码相同也不是没有可能。在不确定的情况下，解决方案是创建一个人造主键。你可以随便定义这样的列，将其添加到表中的唯一目的是用作主键。添加这种列的优点是，可确保它符合前述所有标准。在表中添加这样的列后，将其指定为主键，这就大功告成了！就这么简单。图 2-16 显示了包含人造主键 EmployeeID 后的 Employees 表。

Employees

EmployeeID	EmpLastName	EmpFirstName	EmpCity	EmpState	EmpZip	<< 其他列 >>
98001	Gehring	Darren	Chico	CA	95926	…
98002	Kennedy	John	Portland	OR	97208	…
98003	Thompson	Sarah	Redmond	WA	98052	…
98004	Wilson	Jim	Salem	OR		…
98005	Seidel	Manuela	Medford	OR	97501	…
98006	Smith	David	Fremont	CA	94538	…
98007	Patterson	Neil	SanDiego	CA	92199	…
98008	Viescas	Michael	Redmond	WA	98052	…
98009	Viescas	David	Portland	OR	97207	…

图 2-16　包含新增人造主键的 Employees 表

❖ **说明**　虽然人造主键是一种解决上述问题的简单途径，但无法真正保证表中不会出现重复的数据。例如，如果有人给员工 John Kennedy 添加了新行，并提供了独一无二的人造 EmployeeID 值，你如何知道这个 John Kennedy 与表中已有的员工 98002 不是同一个人呢？

答案是在应用程序代码中添加一个验证过程，以检查出潜在的重复姓名并向用户发出警告。在很多数据库系统中，都可以触发器的方式编写这样的验证代码。每当你修改、添加或删除行时，数据库系统都会自动运行相应的触发器。然而，讨论触发器远远超出了本书的范围，相关细节请参阅数据库系统文档。

至此，强化和微调表结构的全部工作都完成了。下面来看看如何确保所有的表关系都是合理的。

2.5　建立合理的关系

第 1 章说过，如果有两个表，其中一个表中的行以某种方式关联到了另一个表中的行，这两个表之间就存在关系。该章还说过，关系本身可指定为 3 种类型之一：一对一、一对多和多对多；另外，每种关系都是以特定的方式建立的。下面来简单地复习一下。

> ❖ **说明**　本节使用的图示符号源自 Mike Hernandez 在其著作 *Database Design for Mere Mortals* 第 3 版中介绍的图示方法。PK 表示主键列，FK 表示外键列，而 CPK 表示构成复合主键的列。

- **一对一关系**是通过将主表中的主键作为外键插入到从属表中实现的。这是一种特殊的关系，因为在很多情况下，外键同时作为从属表中的主键。图 2-17 展示了这种关系。

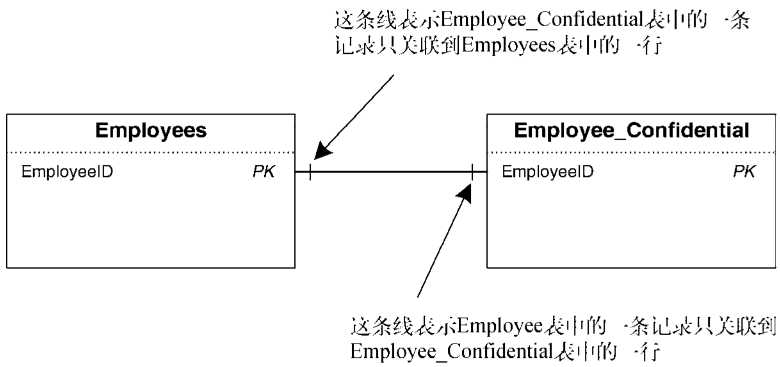


图 2-17　一对一关系

- **一对多关系**是通过将位于“一”端的表的主键作为外键插入到位于“多”端的表中实现的。图 2-18 展示了这种关系。

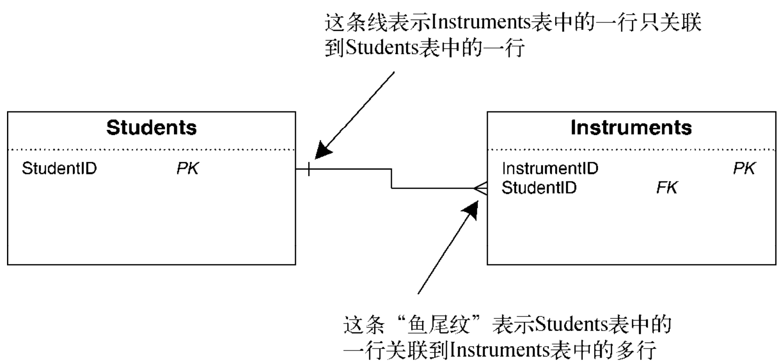


图 2-18　一对多关系

- **多对多关系**是通过创建链接表建立的。为定义链接表，可复制关系涉及的每个表的主键，并用它们构成新表的结构。这些列通常扮演两个角色：它们一道构成链接表的复合主键，同时分别充当外键。图 2-19 展示了这种关系。

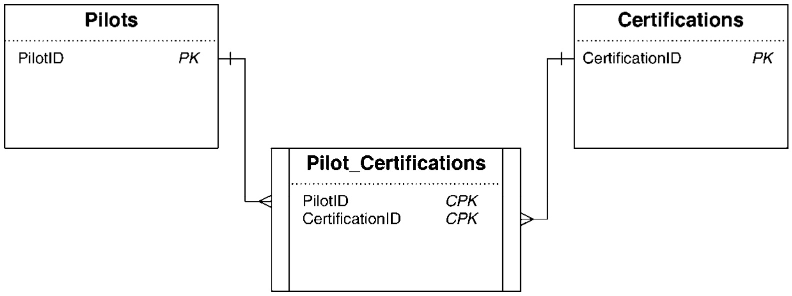


图 2-19　多对多关系

多对多关系都是通过**链接表**来建立的。在这个示例中，链接表为 Pilot_Certifications。一位飞行员可以有任意数量的驾驶证，而一种驾驶证可关联到任意数量的飞行员。

为确保数据库中表之间的关系是牢靠的，必须确定每个关系的特征。你定义的特征指出了删除行时将发生的情况以及表在关系中的参与类型和参与程度。

需要指出的是，用于将两个表关联起来的列的数据类型必须相同，这很重要。例如，对于数据类型为 Int（整型）的主键，只能将其关联到数据类型也是 Int 的外键。不能将数字关联到字符或日期。对于这条规则，唯一的例外是主键是由数据库自动生成的。根据你使用的数据库系统，这种主键被称为 AutoNumber、Identity、Serial 或 Auto_Increment。对于这些类型的主键，都有相应的底层数值数据类型［在 Microsoft Access 中为长整型（Long Integer），在其他大多数数据库系统中为 Int］，因此完全可以将自动生成的主键关联到数据类型为相应底层数据类型的列。也就是说，在 Microsoft Access 中，可将 AutoNumber 主键关联到 Number/Long Integer 外键，而在 Microsoft SQL Server 中，可将 Identity 主键关联到数据类型为 Int 的列。

讨论关系的特征前，必须先声明一点：我将在一个通用而合乎逻辑的参考框架内阐述这些特征。这些特征很重要，因为它们能够确保关系完整性（在有些数据库系统中称为**引用完整性**）。然而，这些特征的实现方式随数据库软件程序而异。要确定这些特征是否得到了支持以及如何实现它们，请参阅你使用的数据库软件的文档。

2.5.1　指定删除规则

删除规则指定了在用户请求从一对一关系中的主表或一对多关系中位于“一”端的表中删除一行时，将发生什么事情。通过指定这种规则，可避免出现孤行（所谓**孤行**，指的是在一对一关系的从属表中，没有关联到主表行的行，或者在一对多关系的位于“多”端的表中，没有关联到“一”端表中行的行）。

可为关系指定两类删除规则：**限制**和**级联**。

- ❑ **限制删除规则**禁止删除这样的行，即它关联到了从属表（一对一关系中）或“多”端表（一对多关系中）的行。要删除行，必须先将与之相关联的行删除。通常都使用这种删除规则。在允许定义删除规则的数据库系统中，限制通常都是默认删除规则，且有时只提供这个选项。
- ❑ 执行的是**级联删除规则**时，如果删除位于关系“一”端的行，将导致系统自动删除从属表（一对一关系）或“多”端表（一对多关系）中所有与之相关联的行。请慎用这种规则，否则可能删除原本想要保留的行！并非所有的数据库系统都支持级联删除。

无论在什么情况下，都务必仔细研究你面临的关系，以确定使用哪种删除规则更合适。可通过回答一个非常简单的问题来确定该使用哪种删除规则。先选择两个表，再自问如下问题：如果［主表或“一”端表名称］中的一行被删除，是否也应该删除［从属表或“多”端表的名称］中相关联的行？

这个问题是通用的，旨在让你能够理解其背后的假设。要使用这个问题，请将方括号中的内容替换为表名。最终的问题类似于这样：如果 Committees 表中的一行被删除，是否也应该删除 Committee_Members 表中相关联的行？

如果这个问题的答案是否定的，就使用限制删除规则，否则就使用级联删除规则。归根结底，这个问题的答案在很大程度上取决于你如何使用存储在数据库中的数据。这就是你必须仔细研究关系并确保选择合适删除规则的原因。

图 2-20 演示了如何图示关系的删除规则。请注意，使用（R）表示限制删除规则，使用（C）表示级联删除规则。

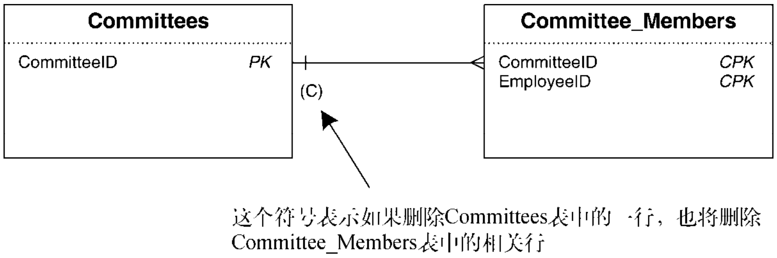


图 2-20 图示 Committees 表和 Committee_Members 表的删除规则

2.5.2 指定参与类型

在两个表之间建立关系时，每个表都将以特定的方式参与其中。给表指定的**参与类型**决定了是否该表必须至少有一行，才能在另一个表中输入数据。参与类型有如下两种。

- **强制**：当前表必须至少有一行才能在另一个表中输入内容。
- **可选**：当前表无须包含任何内容就可在另一个表中输入内容。

在很大程度上，给一对表选择什么样的参与类型取决于组织的业务逻辑。例如，假设贵公司是一家大型公司，由多个部门组成；再假设在你为公司创建的数据库中，包含一个 Employees 表、一个 Departments 表和一个 Department_Employees 表。有关员工的信息都存储在 Employees 表中，有关部门的信息都存储在 Departments 表中，而 Department_Employees 是一个链接表，让你能够将任意数量的部门关联到给定的员工。图 2-21 显示了这些表（在该图中，我使用简单的箭头来指向关系的"多"端）。

Employees

EmployeeID	EmpLastName	EmpFirstName	EmpCity	<< 其他列 >>
7004	Gehring	Darren	Chico	...
7005	Kennedy	John	Portland	...
7006	Thompson	Sarah	Lubbock	...
7007	Wilson	Jim	Salem	...
7008	Seidel	Manuela	Medford	...
7009	Smith	David	Fremont	...
7010	Patterson	Neil	San Diego	...
7011	Viescas	Michael	Redmond	...

Departments

DepartmentID	DepartmentName	Floor
1000	Accounting	5
1001	Administration	5
1002	HumanResources	7
1003	InformationServices	6
1004	Legal	7

Department_Employees

EmployeeID	DepartmentID	Position
7004	1000	Head
7005	1000	Floater
7005	1001	Floater
7007	1001	Staff
7008	1001	Head
7009	1003	Floater
7010	1002	Head
7011	1004	Head

图 2-21 Employees 表、Departments 表和 Department_Employees 表

在最近一次员工会议上，你被要求将一些员工分配到新成立的研发（Research and Development）部。现在的问题是，你想确保在 Departments 表中添加这个新部门后，才能在 Department_Employees 表中将员工分配到该部门。这正是参与类型的用武之地：将 Departments 表的参与类型设置为强制，并将 Department_Employees 表的参与类型设置为可选。通过指定这些设置，可确保仅当一个部门出现在 Departments 表中后，才能在 Department_Employees 表中将员工分配到这个部门。

与删除规则一样，请仔细研究每个关系，确定为其涉及的每个表指定哪种参与类型是合适的。图 2-22 演示了如何图示参与类型。

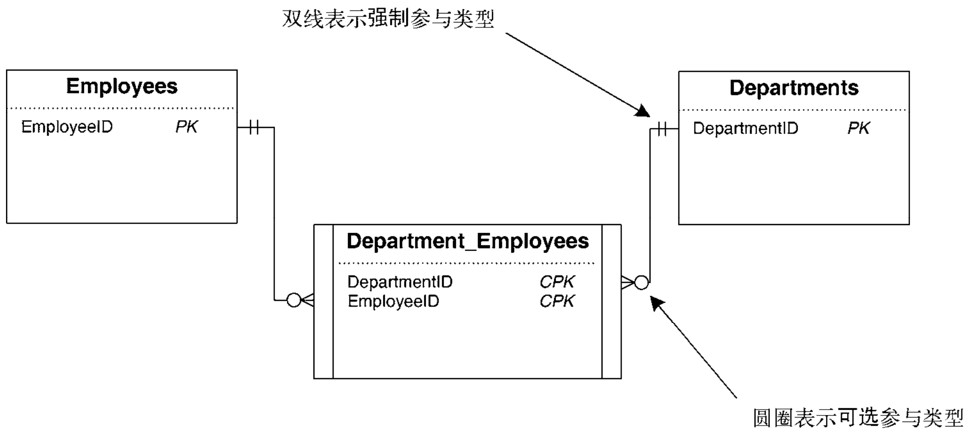


图 2-22　图示 Departments 表和 Department_Employees 表的参与类型

2.5.3　指定参与程度

确定每个表该如何参与关系后，必须确定它们的参与程度。为此，可确定表中的一行最多和最少可关联到另一个表中的多少行，这个过程称为确定表的**参与程度**。参与程度用两个放在括号内并由逗号分隔的数字表示，其中第一个数字表示最少必须关联到多少行，而第二个数字表示最多可关联到多少行。例如，参数程度“(1, 12)”表示最少必须关联到 1 行，最多可关联到 12 行。

在很大程度上说，如何给数据库中的各个表指定参与程度取决于组织查看和使用数据的方式。假设你是一家经纪公司的经纪人，而该公司的数据库中有两个表——Agents 和 Entertainers。再假设这两个表之间存在一对多关系：Agents 表中的一行可关联到 Entertainers 表中的多行，但 Entertainers 表中的一行只能关联到 Agents 表中的多行。在这里，我确保了一位艺人只能有一位经纪人（这完全杜绝了艺人在不同经纪人之间玩手腕，实乃好事一件）。

几乎在所有情况下，关系中“多”端的最大可关联行数都是无限的，但在有些情况下，业务规则可能要求对参与程度进行限制，如对注册一门课程的学生数量进行限制。在这里的示例中，假设老板想确保所有经纪人都有平等的机会获得高佣金，并最大限度地减少经纪人之间的勾心斗角。为此，他制定了一个新政策，规定每个经纪人最多可代理 6 位艺人（虽然他认为长远看这行不通，但还是想试试）。为实现这种新政策，他按下面这样指定了这两个表的参与程度。

Agents	(1, 1) ——每位艺人只能关联到一位经纪人
Entertainers	(0, 6) ——虽然经纪人并非必须与艺人相关联，但最多只能与6位艺人相关联

图 2-23 演示了如何图示这两个表的参与程度。

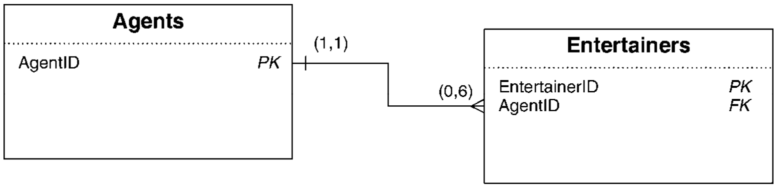


图 2-23 图示 Agents 表和 Entertainers 表的参与程度

指定参与程度后，该决定要让数据库系统如何实现关系了。如何选择取决于数据库系统提供了哪些功能。大多数数据库系统支持的最简单的实现方式是，对“多”端表中外键的取值进行限制，禁止用户输入相关的“一”端表中没有的值。为指出这一点，可在指向“一”端的关系线旁边放置字母 R，并将其用括号括起，如图 2-24 所示。

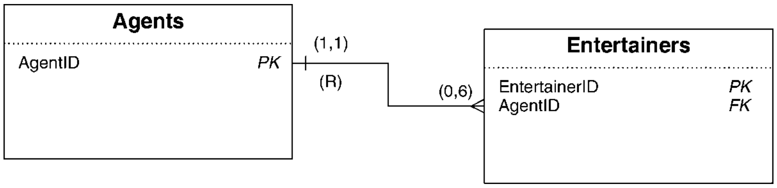


图 2-24 图示 Agents 表和 Entertainers 表的所有关系的特征

在有些数据库系统中，可以定义一个规则，在用户修改“一”端表中主键的值时，将主键值从“一”端级联（C）到“多”端表。从本质上说，就是当你修改“一”端表中主键的值时，数据库系统将相应地修改“多”端表中相关联行的外键值。另外，有些数据库系统提供了这样的功能：当你删除“一”端表中的一行时，它将自动删除（D）“多”端表中相关的行。相关细节，请参阅你使用的数据库系统的文档。

❖ **说明** 要实施参与程度约束，必须在数据库定义中定义一个或多个触发器或约束（如果数据库系统支持这些功能的话）。

2.6 就这些吗

通过使用本章介绍的方法，就在通过确保数据库基本的数据完整性方面迈出了第一步。下一步是着手研究组织查看和使用数据的方式，以便能够为数据库制定并实施业务规则。但要充分利用数据库，应重新回到起点，基于良好的设计方法重走一遍数据库设计过程。遗憾的是这些主题超出了本书的范围，但你可通过阅读如下著作来学习良好的设计方法：Michael J. Hernandez 编著的 *Database Design for Mere Mortals* 第 3 版；Thomas Connolly 和 Carolyn Begg 编著的 *Database Systems: A Practical Approach to Design, Implementation, and Management* 第 6 版。需要牢记的要点是，数据库结构越可靠，从数据库提取信息以及为其创建应用程序就越容易。

2.7 小结

本章首先简要地讨论了为何要关心数据库结构是否合理，说明了设计糟糕的表可能带来数不胜数的问题，其中最突出的是数据完整性问题。

接下来讨论了如何微调每个表包含的列，并且阐释了给列指定合适的名称非常重要，因为这将确保每个名称都是有意义的，还可帮助你发现与列结构相关的潜在问题。现在，你知道了如何微调列结构，以确保它们符合一些简单的规则。这些规则可用于处理一些问题，如确保每列都表示表所表示的客体的一个特征，只包含单个值，绝不会存储计算结果。此外还讨论了多部分列和多值列带来的问题，以及如何妥善地解决这些问题。

之后讨论了如何微调表。你了解到，鉴于众多相同的原因，表名和列名一样也很重要。你知道了如何给表指定有

意义的名称，以及如何确保每个表都只表示一种客体。然后讨论了一组规则，可用它们来确保每个表的结构都是合理的。虽然有些规则看起来要求你重复微调列结构时所做的工作，但用于微调表结构的规则实际上增加了一层保险，可确保表结构尽可能合理。

接下来讨论了主键。你学习了给数据库中的每个表指定主键的重要性。你现在知道，主键必须具备一系列特征，因此必须非常仔细地为每个表选择充当主键的列。你还了解到，如果表中没有具备所有主键特征的列，可创建一个人造主键。

最后讨论了如何建立合理的关系。在复习了三种关系之后介绍了如何图示它们。接着学习了如何为关系指定删除规则，并使用图示表示出来。删除规则非常重要，因为它有助于避免孤行。本章讨论的最后两个主题是关系涉及的表的参与类型和参与程度，你了解到参与类型可以是强制的，也可以是可选的；另外，还可指定表中一行可关联到另一个表中行数的范围。

下一章将简单地介绍 SQL 的历史以及它是如何发展成为当前版本（SQL:2016）的。

第3章

SQL 简史

“宗教只有一种，虽然其形式多样。”

——萧伯纳，*Plays Pleasant and Unpleasant*

本章涵盖如下主题：

- SQL 的起源
- 早期的厂商实现
- 标准应运而生
- ANSI/ISO 标准的发展历程
- 商用实现
- 展望未来
- 为何要学习 SQL
- 本书基于哪个 SQL 版本
- 小结

谈历史总离不开那些似是而非的意外事件、政治阴谋和人性弱点，从这种意义上说，SQL 的历史也不例外。自关系模型面世后，SQL 就以这样或那样的形式出现了。关于 SQL 悠久的沉浮历史，有不少详细的叙述，但本章将详细说说这种数据库语言的起源、发展历程和未来之路。本章的目标有两个：一是让你了解 SQL 是如何逐渐成熟，成为当今大多数关系型数据库系统使用的语言的；二是明白为何学习怎样使用 SQL 很重要。

3.1 SQL 的起源

第 1 章说过，20 世纪 70 年代，E. F. Codd 博士推出了关系型数据库模型。这个具有里程碑意义的事件发生后不久，诸如大学和研究实验室等组织开始致力于开发一种语言，并将其作为支持关系模型的数据库系统的基础。基于最初所做的工作，多种语言于 20 世纪 70 年代早中期应运而生，随后又推出了当前还在使用的 SQL 和基于 SQL 的数据库。但 SQL 到底是怎么来的呢？它的发展历程是什么样的呢？未来又将如何？要回答这些问题，必须从发生在 IBM 圣塔特蕾莎研究实验室的故事讲起。

20 世纪 70 年代初，IBM 推出了一个重大项目——System R，旨在证明关系模型的可行性，同时获得一些设计和实现关系型数据库的经验。研究人员于 1974 年和 1975 年竭尽所能，推出了一个最简单的关系型数据库原型，事实证明这种努力是成功的。

除了致力于开发一种可行的关系型数据库外，研究人员还定义了一种数据库语言。在最初为定义这种语言的各种尝试中，圣塔特蕾莎研究实验室所做的努力在商业意义上无疑是最为重大的。1974 年，Donald Chamberlin 博士与同事一道推出了英语式结构化查询语言（Structured English Query Language，SEQUEL），让用户能够使用定义明确的英语式句子查询关系型数据库。Chamberlin 博士与同事首先在原型数据库 SEQUEL-XRM 中实现了这种语言。

SEQUEL-XRM 获得的反馈很不错，并获得了巨大的成功，这激励着 Chamberlin 博士与同事继续这方面的研究。

在 1976 年到 1977 年，他们对 SEQUEL 做了全面修订，并将修订后的版本命名为 SEQUEL/2。然而，由于已经有人使用了缩略语 SEQUEL，出于法律方面的原因，他们不得不将 SEQUEL 改名为 SQL（结构化查询语言或 SQL 查询语言）。直到现在，很多人依然将 SQL 读为 sequel，虽然被广泛接受的“官方”发音是 es-cue-el。最初的 SQL 提供了多种新功能，如支持多表查询以及多名用户访问共享数据。

SQL 面世后不久，IBM 发起了一个更为雄心勃勃的新项目，旨在开发一个原型数据库，进一步证明关系模型的可行性。这个新原型被命名为 System R，它基于 SQL 的很大一部分。初步开发工作基本完成后，IBM 将 System R 安装到了很多内部场点和客户场点，对其进行测试和评估，并根据这些用户的使用体验和反馈对其做了大量的修改。1979 年，IBM 结束了这个项目，但得出的结论是关系模型确实是一项可行的数据库技术，具备极大的商业潜力。

❖ **说明** 这个项目的另一个重要成功之处，是推动了 SQL 的发展。实际上，SQL 基于一门名为“以关系表达式的方式指定查询”（Specifying Queries As Relational Expressions，SQUARE）的研究语言，该研究语言是 1975 年开发的（早于 System R 项目），旨在使用英语式句子实现关系代数。

你很可能会问：既然 IBM 得出了具有商业潜力的结论，为何要结束这个项目呢？20 世纪 70 年代末，我观看了 System R 的演示过程，它有很多亮点，但受制于当时的硬件技术，即便是简单的查询也需要几分钟才能运行完毕。显然，它颇具潜力，但要吸引企业，必须有更好的硬件和软件。

3.2 早期的厂商实现

由于 IBM 研究实验室于 20 世纪 70 年代所做的工作，各种技术期刊对关系模型萌发了强烈的兴趣，数据库技术研讨会也对其优劣进行了激烈的讨论。到 20 世纪 70 年代末，大家清楚地意识到，IBM 对关系型数据库技术和 SQL 有强烈的兴趣，并致力于推出基于这种技术的产品。这导致很多厂商都在猜测 IBM 会在多久后推出第一款这样的产品。有些厂商的反应非常快，决定尽早开发这样的产品，而不是等待 IBM 来引领这个市场。

1977 年，多名工程师于美国加州门洛帕克组建了 Relational Software 公司，旨在打造一款基于 SQL 的关系型数据库产品。他们将这款产品命名为 Oracle，并于 1979 年将其推向市场。这是第一款商用关系型数据库管理系统（RDBMS），比 IBM 向市场推出类似产品早了两年。Oracle 的优点之一在于，它运行在 Digital 的 VAX 小型机上，而不是更昂贵的 IBM 大型机上。从那时起，Relational Software 更名为 Oracle，并一直是领先的 RDBMS 软件厂商之一。

与此同时，加州大学伯克利分校计算机实验室的 Michael Stonebraker、Eugene Wong 等几位教授也加入了研究关系型数据库技术的行列。与 IBM 的研究小组一样，他们也开发了一个关系型数据库原型，并将这款产品命名为 Ingres。Ingres 包含一门名为“查询语言”（Query Language，QUEL）的数据库语言，该语言类似于 SQL，但结构化程度更高，对英语式句子的使用更少。随着 SQL 作为标准数据库语言变得越来越明朗，Ingres 最终变成了一款基于 SQL 的 RDBMS。20 世纪 80 年代，多位教授离开加州大学伯克利分校，组建了 Relational Technology 公司，并于 1981 年发布了第一个商用的 Ingres 版本。Relational Technology 经历了多次转型，当前为 Computer Associates International 的子公司，而 Ingres 现为 Actian 公司所有并提供支持，当前依然是行业领先的数据库产品。

再回过头来说说 IBM。1981 年，IBM 发布了其 RDBMS 产品 SQL/Data System（SQL/DS），并于 1982 年将其推向市场。1983 年，该公司推出了用于 VM/CMS 操作系统（IBM 推出的多种大型机操作系统之一）的 SQL/DS 新版本，并发布了一款新的 RDBMS 产品——Database 2（DB2），用于运行 IBM 主流操作系统 MVS 的大型机。DB2 于 1985 年推向市场，是 IBM 最重要的 RDBMS 产品，其技术被嵌入到整个 IBM 产品线中。顺便说一句，IBM 没变，还是原来的 IBM。

在接下来的 40 多年中，最初作为研究项目的 System R 影响了商业的各个方面，并发展成了一个数十亿美元的行业。

3.3 标准应运而生

在数据库语言开发如火如荼的过程中，是否有人想过要进行标准化呢？虽然这种想法在数据库界反复出现过，但

在该由谁来制定标准以及标准应基于哪种方言方面，从未达成一致。有鉴于此，每家厂商都在不断地开发和改进自己的数据库产品，期望其数据库产品及其使用的 SQL 成为行业标准。

基于客户的反馈和需求，很多厂商在其 SQL 方言中添加了独特的元素，于是一个非官方标准应运而生。相比于当前的标准，该规范不过是小儿科，它只包含各种 SQL 方言都有的元素。然而，这个规范确实向客户提供了一组核心准则，让他们能够对市面上的各种数据库做出判断，还提供了一些让用户能够用于不同数据库程序的知识。

鉴于对官方关系型数据库语言标准的需求日益迫切，美国国家标准学会于 1982 年委托 X3 建立了数据库技术委员会 X3H2，以编写关系型数据库语言标准提案。X3 是 ANSI 管理的众多组织之一，而 X3H2 是向 X3 报告的众多技术委员会之一。X3H2 由数据库行业的专家以及来自各个主要 SQL 数据库厂商的代表组成。最初，该委员会对提议的各种语言的优缺点做了审核和讨论，并以 Ingres 使用的数据库语言 QUEL 为基础，着手制定标准。但考虑到市场的力量以及 IBM 在 SQL 上日益增多的投入，该委员会转而决定在 SQL 的基础上制定标准提案。

X3H2 委员会的标准提案主要是基于 IBM DB2 使用的 SQL 方言。在接下来的两年中，该委员会制定了多个标准版本，并对 SQL 做了一些改进。然而，这些改进带来了糟糕的结果：新标准不再与既有的主要 SQL 方言兼容。X3H2 很快意识到，所做的修改对 SQL 的改善程度并不大，不值得为此付出不兼容的代价，因此该委员会决定恢复原来的标准版本。

1986 年，ANSI 批准了 X3H2 提议的标准，将其命名为 ANSI X3.135-1986 Database Language SQL，俗称 SQL/86。虽然在 ANSI 批准前，X3H2 对标准做了一些细微的修订，但 SQL/86 只定义了一组最基本的共同需求，所有的数据库厂商都能遵守。从本质上说，该标准只定义了各种 SQL 方言都包含且很多数据库厂商都实现了的元素，但为 SQL 语言及其实现的进一步发展提供了坚实的基础。

1987 年，国际标准组织（ISO）批准了标准 ISO 9075-1987 Database Language SQL，该标准与 ANSI SQL/86 完全相同（这两个标准现在依然被简称为 SQL/86），这让国际上的其他数据库厂商能够基于美国数据库厂商遵守的标准开展工作。虽然成了官方标准，但那时的 SQL 根本谈不上完善。

3.4　ANSI/ISO 标准的发展历程

SQL/86 发布后不久，就遭到了政府以及 C. J. Date 等行业专家的诟病。这些批评者指出的问题包括：SQL 语法重复（定义同一个查询的方式有多种），不支持有些关系运算符，不能保证引用完整性。虽然在 SQL/86 发布前 X3H2 就发现了这些问题，但该委员会认为，发布一个不完善的标准胜过根本没有标准。

针对有关引用完整性方面的批评，ISO 和 ANSI 都对标准做了修订：1989 年年中，ISO 发布了修订后的标准 ISO 9075:1989 Database Language SQL with Integrity Enhancements；1989 年年末，ANSI 发布了修订后的标准 X3.135-1989 Database Language SQL with Integrity Enhancements，俗称 SQL/89。然而，在这一年，ANSI X3H2 委员会所做的工作还不止这些，它还力图解决政府提出的一个重要问题。

有些政府用户抱怨该标准未明确规定如何在传统编程语言中嵌入 SQL（虽然标准包含这样的规范，但该规范被降格为附件）。这些用户担心厂商可能不支持可移植的嵌入式 SQL 实现，因为标准没有明确要求它们这样做。针对这种抱怨，X3H2 发布了另一个标准——ANSI X3.168-1989 Database Language Embedded SQL，要求必须遵循嵌入规范。有趣的是，ISO 没有因为国际数据库界类似的担心而发布相应的标准，这意味着在发布标准 SQL/92 前，ISO 一直没有制定有关在编程语言中嵌入 SQL 的规范。

SQL/86 和 SQL/89 标准一点也不完善，它们没有定义商业数据库系统需要的一些重要特性。例如，这两个标准都没有指定在定义数据库后修改其结构的方式（包括在数据库系统内部进行修改的方式）。这意味着无法修改或删除结构性组件（如表或列），也无法修改数据库的安全性设置。例如，可使用 CREATE 命令来创建表，但这个标准没有定义用于删除表的 DROP 命令，也没有定义用于修改表的 ALTER 命令。另外，可使用 GRANT 命令授予访问表的权限，但这个标准没有定义用于撤销访问权限的 REVOKE 命令。具有讽刺意味的是，所有基于 SQL 的商用数据库都提供了这些功能，但这两个标准都没有定义它们，因为每家厂商实现它们的方式不同。还有其他很多基于 SQL 的数据库都实现了，但这两个标准都没有定义的功能。同样，这是因为每家厂商实现它们的方式不同。

SQL/89 制定完毕时，ANSI 和 ISO 都已对 SQL 做重大修订，使其成为一种完善而健壮的语言。如果将这个新版本作为标准，它将被命名为 SQL/92，也就定义了大多数主要数据库厂商实现了的功能。但 ANSI 和 ISO 的一个主要目标是，避免再次定义一个“最小公分母”式标准。有鉴于此，它们决定在标准中包含还未被广泛接受的功能，并添加还未被实现的功能。

1992 年 10 月，ANSI 和 ISO 都发布了新的 SQL 标准，分别名为 X3.135-1992 Database Language SQL 和 ISO/IEC 9075:1992 Database Language SQL（这些文档的编撰工作在 1991 年年底就已完成，但一些最后的修订是在 1992 年完成的）。SQL/92 标准文档比 SQL/89 标准文档长得多，涵盖的范围也广得多。例如，它提供了在定义数据库后对其结构进行修改的途径，支持更多操纵字符串及日期和时间的操作，定义了更多的安全功能。相比于以前的标准，SQL/92 向前迈出了很大一步。

所幸标准委员会在一定程度上预料到了这将带来的问题。为了让数据库厂商能够循序渐进地遵守新标准，ANSI 和 ISO 定义了三个等级的 SQL/92。

- ENTRY SQL：类似于 SQL/89，但包含简化从 SQL/89 过渡到 SQL/92 的功能，还有对 SQL/89 标准中的错误进行更正的功能。定义这个等级旨在简化实现，因为其功能在既有产品中已得到广泛实现。
- INTERMEDIATE SQL：这个等级包含新标准 SQL/92 中的大部分功能。两个委员会都基于几个因素决定在这个等级中包含哪些功能，其总体目标是对标准进行改进，让 SQL 能够更好地支持关系模型中的概念，同时对不清晰的语言进行改进。很容易做出判断，应在这个等级中包含已经有厂商实现了且能够满足上述目标的功能。为此，标准委员会优先考虑的是这样的功能，即 SQL 数据库系统用户需要、能够满足上述目标且对大多数厂商来说实现起来比较容易。定义这个等级旨在确保能够改进既有产品，使其实现尽可能健壮。
- FULL SQL：这个等级包含完整的 SQL/92 规范，显然，它包含前两个等级没有的较为复杂的功能。这个等级包含一些这样的功能，即对于满足客户的需求或进一步净化 SQL 语言来说很重要，但对大多数厂商来说，马上实现它们比较困难。遗憾的是，并没有要求必须遵循 FULL SQL，因此需要过一段时间，数据库产品才会完全实现这个标准。

很多数据库厂商继续努力实现 SQL/92 定义的功能，同时开发并实现了其独有的功能。对 SQL 标准所做的增补被称为**扩展**。例如，除 SQL/92 指定的 6 种数据类型外，厂商还可能提供其他数据类型。虽然这些扩展增加了产品的功能，同时让厂商有别于其他厂商，但也存在缺点。添加扩展带来的一个主要问题是，导致每家厂商的 SQL 方言与标准的差别更大，而这将导致数据库开发人员无法创建可移植的应用程序，即使用任何 SQL 数据库都能运行的应用程序。

其他 SQL 标准

当前，ANSI/ISO SQL 标准是最被广泛接受的标准，这意味着还有其他涉及 SQL 的标准。下面是其他一些较为重要的 SQL 标准。

- X/OPEN：一组欧洲数据库厂商（统称为 X/OPEN）制定的一系列标准，旨在帮助营造基于 UNIX 的可移植应用程序环境。在欧洲市场，能够在不做修改的情况下将应用程序从一种计算机系统移植到另一种计算机系统很重要。虽然 X/OPEN 成员在这组标准中包含了 SQL 标准，但其版本在很多方面都有别于 ANSI/ISO 标准。
- SAA：IBM 一直在开发自己的 SQL 方言，并将其包含在系统应用程序架构（Systems Application Architecture，SAA）规范中。SAA 规范的目标之一是，将 IBM 的 SQL 方言集成到 IBM 的整个数据产品线中。虽然这个目标从未实现，但 SQL 依然在统一 IBM 数据库产品方面扮演着重要角色。
- FIPS：1987 年，SQL 标准被美国国家标准技术研究所（NIST）批准为联邦信息处理标准（Federal Information Processing Standard，FIPS）。该标准最初名为 FIPS PUB 127，指定了 RDBMS 必须遵守哪个等级的 ANSI/ISO 标准。从那时起，美国政府机构使用的所有关系型数据库产品都必须遵守最新的 FIPS 标准。
- ODBC：1989 年，一组数据库厂商建立了 SQL Access 集团，旨在解决数据库互操作性问题。虽然这些厂商最初所做的努力不太成功，但随后它们扩大了目标范围，力图制定一种将 SQL 数据库绑定到用户界面语言的标准。1992 年，这种努力有了结晶，那就是调用级接口（Call-Level Interface，CLI）规范。在同一年，微软发布

了基于 CLI 标准的开放数据库连接（Open Database Connectivity，ODBC）规范。从那时起，ODBC 就成了访问 SQL 数据库以及在 SQL 数据库之间共享数据的事实标准。

随着 ANSI/ISO SQL 新版本的推出，这些标准也在不断向前发展，而有时候它们也会独立于 ANSI/ISO SQL 向前发展。

1997 年，ANSI 下属组织 X3 更名为美国信息技术标准委员会（NCITS），而负责制定 SQL 标准的技术委员会更名为 ANSI NCITS-H2，最近该委员会又更名为 INCITS DM32.2。鉴于 SQL 标准越来越复杂，ANSI 和 ISO 标准委员会在着手制定 SQL3 标准（这样命名是因为这是对标准做的第 3 次重大修订）时，一致同意将这个标准分成 12 个部分和一个附录，以便能够同时处理各个部分。从 1997 年起，又定义了另外两部分。

表 3-1 列出了 SQL:2016（ISO/IEC 9075:2016）标准的各个部分，对其做了描述并指出了它们的状态。

表 3-1 SQL 标准的组成部分

名　称	状　态	描　述	在 SQL:2016 中的页数
Part 1: Framework（SQL/Framework）	完成于 1999 年，2003 年、2008 年、2011 年和 2016 年做了修订	描述标准的各个部分，包含适用于所有部分的信息	78
Part 2: Foundation（SQL/Foundation）	1992 标准的核心，1999 年、2003 年、2008 年、2011 年和 2016 年做了修订	定义了 SQL 语言的数据定义和数据操作部分的语法和语义	1707
SQL/OLAP（Online Analytical Processing）	1999年被合并到Foundation部分	描述用于分析处理的函数和操作，是 SQL/Foundation 部分的一个附录	
Part 3: Call-Level Interface（SQL/CLI）	完成于 1995 年，1999 年、2003 年、2008 年和 2016 年做了扩展	由 SQL Access 集团开发，对应于微软的 ODBC 规范	391
Part 4: Persistent Stored Modules（SQL/PSM）	完成于 1996 年。1999 年，存储例程和 CALL 语句被移到 Foundation 部分，其他内容于 2003 年、2011 年和 2016 年做了修订	定义在用户定义的函数和过程中很有用的过程型语言 SQL 语句。对存储过程、存储函数、CALL 语句和例程调用的支持最终移到了 SQL/Foundation 部分	188
Part 5: SQL/ Bindings	1999 年，SQL 嵌入规范放在了一个独立的部分中；2003 年，这部分被并入到 Foundation 部分	指定在非对象编程语言中如何嵌入 SQL；在下一版本的 SQL 标准中，这部分将合并到 SQL/Foundation 中	
Part 6: Transaction（XA Specialization）	1999 年被撤销	专门针对 SQL 的 X/OPEN XA 规范	
Part 7: SQL/ Temporal	2003 年被撤销	定义了如何存储和检索时序数据。在有关时序数据的需求和细节方面，存在不同的看法，因此最近几年这方面的工作暂停了	
Part 8: SQL/ Objects Extended Objects	1999年被合并到Foundation部分	指定 RDBMS 如何处理应用程序定义的抽象数据类型	
Part 9: Management of External Data（SQL/MED）	ISO 版本完成于 2003 年，并于 2008 年和 2016 年做了修订	指定 SQL/Foundation 中没有的语法和定义，让 SQL 能够访问非 SQL 数据源（文件）	471
Part 10: Object Language Bindings（SQL/OLB）	最初仅包含于 ANSI 标准中，完成于 1998 年；ISO 于 1999 年、2003 年、2008 年和 2016 对其做了修订	指定在 Java 编程语言中嵌入 SQL 的语法和语义，对应于 ANSI 标准 SQLJ Part 0	376
Part 11: Information and Definition Schemas（SQL/Schemata）	从 Foundation 部分提取出来的，完成于 2003 年，并于 2008 年和 2016 年做了修订	信息和定义模式（schema）	327

（续）

名　称	状　态	描　述	在 SQL:2016 中的页数
Part 12: SQL/ Replication	开始于 2000 年的一个项目，由于进度缓慢于 2003 年终止	定义如何支持和简化SQL数据库复制	
Part 13: SQL Routines and Types Using the Java Programming Language（SQL/ JRT）	最初只包含在 ANSI 标准中，基于 SQL/92，完成于 1999 年；2003 年、2008 年和 2016 年作为国际标准做了修订	定义如何在 SQL 数据库中使用 Java 代码	151
Part 14: XMLRelated Specifications（SQL/XML）	完成于 2003 年，并于 2006 年、2008 年、2011 年和 2016 年进行了扩充	定义如何在 SQL 数据库中使用 XML，对应于规范 W3C XQuery V1.1	444

3.5　商用实现

本章前面说过，SQL 最初出现在大型机环境中。自 1979 年起，DB2、Ingres 和 Oracle 等产品就相继面世，并将 SQL 作为处理关系型数据库的首选方法。20 世纪 80 年代，关系型数据库得以进入个人计算机领域，诸如 R:BASE、dBase IV 和 Super Base 等产品让数据库表中的数据对用户来说触手可及。然而，直到 20 世纪 80 年代末和 90 年代初，SQL 才成为桌面关系型数据库使用的语言，而首先打破这种僵局的无疑是 1992 年推出的 Microsoft Access 第 1 版。

20 世纪 90 年代初，客户/服务器计算横空出世，诸如 Microsoft SQL Server 和 Informix-SE 等 RDBMS 程序被设计成能够在各种多用户环境中向用户提供数据库服务。从 2000 年起，让用户能够通过互联网访问数据库信息已成为风尚。很多企业都接受了电子商务的理念，那些还未建立网站的企业都开始迅速建立网站。这导致数据库开发人员希望有功能更强大的客户/服务器数据库，还有久负盛名的大型机 RDBMS 产品的新版本，以便能够使用它们来开发和维护网站所需的数据库。为满足这种需求，一种办法是将数据迁移到云端，以便使用服务器通过互联网来分享数据。亚马逊、微软和 IBM 都推出了流行的云服务，正如你预期的，几乎所有这些云服务器使用的都是支持 SQL 的数据库。

我很想将支持 SQL 的主流产品一一列出，但如果这样做，将需要很多页。一言以蔽之，SQL 已风行于商用数据库系统中。

3.6　展望未来

1999 年，我与 Mike Hernandez 合作编写了本书的第 1 版，那时标准委员会刚完成对 SQL3 的最后修订——这一天可谓姗姗来迟。从那时起，又依次发布了 SQL:1999、SQL:2003、SQL:2008、SQL:2011 和 SQL:2016。2017 年年初，ANSI 和 ISO 委员会正努力工作，致力于制定另一个标准——SQL/MM（多媒体）。这个标准由 5 部分组成：Framework、Full Text、Spatial、Still Image 和 Data Mining。1986 年，这两个标准委员会的脚步远远落后于商用实现，但现在完全可以说 SQL 标准在功能方面早已跟上既有数据库系统的步伐，在很多方面还走在了它们前面。

3.7　为何要学习 SQL

通过学习 SQL，可获得从关系型数据库中检索信息所需的技能，还可帮助你理解很多 RDBMS 产品提供的图形查询界面背后的机制。搞明白 SQL 后，你就能编写复杂的查询，并具备必要的知识，能够在出现问题时排除查询中存在的故障。

鉴于 SQL 被用于众多不同的 RDBMS 产品中，你学到的技能在各种平台中都有用武之地。例如，学习一种产品（如 Microsoft Access）使用的 SQL 后，即便公司转而使用 Microsoft SQL Server、Oracle 或 IBM DB/2，你学到的知识也能派上用场。你无须重新学习 SQL，而只需了解你学过的 SQL 方言与当前产品使用的 SQL 方言之间的不同之处。这就像你学习了英式英语，那么当你移居到美国后，只需将有些单词中的字母 u 删除即可（例如，是 favor 而不是 favour）。

再说一次，SQL 已成为风尚。很多厂商投入了大量的资金和时间做研究工作，旨在在其 RDBMS 产品中支持 SQL；同时大量企业和组织的信息技术基础设施都建立在 RDBMS 产品的基础之上。基于在本章学到的知识，你可能猜到了，SQL 将继续发展以满足市场的需求。

3.8　本书基于哪个 SQL 版本

这个问题问得好！别忘了，本书是为普通人编写的。最新的 SQL 标准文档长达 5000 页，因此本书不可能面面俱到。我将力图让你牢固地掌握几乎每个商用实现都支持的基本功能，即 Framework 和 Foundation 部分定义的功能。另外，我还将通过 4 种最流行的实现提供示例数据库和本书所有问题的解决方案，这 4 种实现分别是 Microsoft Office Access 2016、Microsoft SQL Server 2016、MySQL 5.7 Community Edition 和 PostgreSQL 9.6。当你学习一门语言，现在时、过去时和将来时都是基础；如果学习虚拟语气、过去完成时或进行时，就得参考更高阶的图书。

3.9　小结

本章首先讨论了 SQL 的起源。SQL 是在关系模型推出后不久开发的一种关系型数据库语言，它的早期发展历程与关系模型本身的发展历程紧密相关。

接着讨论了各个数据库厂商最初提供的关系模型实现。你了解到第一个关系型数据库是在大型机中实现的，还有 IBM 和 Oracle 是如何成为数据库行业巨擘的。

然后讨论了 ANSI SQL 标准的起源。你了解到在 ANSI 决定制定官方标准前，存在一个非官方标准。我们讨论了 ANSI X3H2 委员会最初所做的规范工作，并指出新标准虽然只包含所有产品都提供了的功能，但为 SQL 的继续发展打下了基础。你还了解到，ISO 也发布了自己的标准，它与 ANSI 规范完全一致。

紧接着讨论了 ANSI/ISO 标准的发展历程，你了解到最初的标准遭到了各色人员和组织的诟病。然后讨论了面对诟病，ANSI/ISO 对标准做了多次修订；各个版本鱼贯而出，最终推出的 SQL/92 定义了多个合规等级，让厂商能够循序渐进地实现这个标准指定的功能。接下来讨论了 SQL 标准在 1992 年后的进展情况，并简要地介绍了商用 SQL 数据库的发展历程。

最后简要地展望了 SQL 的未来。你了解到 SQL:2016 是一个比 SQL/92 复杂得多的标准。此外，本章还解释了 SQL 将继续发展的原因，提供了一些学习 SQL 的理由，并指出了本书将涵盖 SQL 标准的哪些部分。

第二部分

SQL 基础

第 4 章

创建简单查询

“像智者一样思考，但用通俗的语言交流。”

——威廉·巴特勒·叶芝

本章涵盖如下主题：

- SELECT 简介
- SELECT 语句
- 说点题外话：数据和信息
- 将请求转换为 SQL
- 消除重复行
- 对信息进行排序
- 保存所做的工作
- 语句举例
- 小结
- 练习

对 SQL 的历史有了大致了解后，该来学习这种语言本身了。前言中说过，本书的大部分篇幅都将用于介绍 SQL 的数据操作部分，因此我将首先专注于 SQL 的主力——SELECT 语句。

4.1 SELECT 简介

SELECT 的地位在其他所有关键字之上，是 SQL 的核心。它是构建复杂、功能强大的 SQL 语句的基石，也是从数据库表中检索信息的途径。通过将 SELECT 同其他关键字和子句结合起来使用，可以以数不胜数的方式查找和查看信息。几乎任何与“谁”“什么”“什么地方”“什么时候”乃至“如果……将会怎样”“多少”相关的问题，都可使用 SELECT 来回答。只要妥善地设计了数据库并收集了合适的数据，你就能获得所需的答案，为组织做出合理的决策。阅读到第五部分时，你会发现，你将使用学到的 SELECT 技巧来创建 UPDATE、INSERT 和 DELETE 语句。

在 SQL 中，SELECT 操作可分解为 3 个更小的操作，我将这些操作称为 SELECT 语句、SELECT 表达式和 SELECT 查询（通过以这种方式分解 SELECT 操作，你能轻松地明白其复杂性）。在这些操作中，每个都有自己的关键字和子句，让你能够灵活地创建 SQL 语句来回答问题。在本章后面你将看到，我们甚至可以以各种方式组合这些操作，以回答非常复杂的问题。

本章将首先讨论 SELECT 语句，并简要地介绍一下 SELECT 查询。第 5 章和第 6 章将更深入地研究 SELECT 语句。

❖ **说明**　在有些介绍关系型数据库的图书中，可能使用术语**关系**来表示**表**，使用**元组**或**记录**来表示**行**，使用**属性**或**字段**来表示**列**。然而，SQL 标准做了规定，要求使用术语**表**、**行**和**列**来表示上述数据库元素。在本书余下的篇幅中，我将遵守 SQL 标准，始终使用这 3 个术语。

4.2 SELECT 语句

SELECT 语句是回答与数据库相关问题的基石。当你创建并执行 SELECT 语句时，就是在查询数据库（我知道这一点显而易见，这里指出来旨在让每位读者都位于同样的起点）。实际上，在很多 RDBMS 程序中，都可将 SELECT 语句保存为**查询**、**视图**、**函数**或**存储过程**。听到有人说要查询数据库时，你就知道他将执行某种形式的 SELECT 语句。根据使用的 RDBMS 程序，你可在命令行中直接执行 SELECT 语句，可在交互式示例查询（Query by Example，QBE）网格中执行，或在代码块中执行。无论选择以什么样的方式来定义和执行 SELECT 语句，语法都是相同的。

> ❖ **说明** 很多数据库系统都提供了对 SQL 标准的扩展，让你能够在函数和存储过程中编写复杂的编程语句（如 If…Then…Else），但具体的语法因产品而异。介绍这些编程语言（如 Microsoft SQL Server 的 Transact-SQL 或 Oracle 的 PL/SQL）不在本书的范围之内，但第 19 章将介绍 SQL 标准中定义的 If…Then…Else [CASE]基本形式。在特定的数据库系统中，你将使用作为基石的 SELECT 语句来创建函数和存储过程。本书将使用术语**视图**来表示保存的 SQL 语句，即便这些 SQL 语句被嵌入在函数或过程中。

SELECT 语句由多个关键字组成，这些关键字称为**子句**。为定义 SELECT 语句，可以以不同的方式配置这些子句，来获取所需的信息。在这些子句中，有些是必不可少的，而其他的是可选的。另外，每个子句都包含一个或多个关键字，用于指定必不可少或可选的值；子句根据这些值来获取 SELECT 语句请求的信息。图 4-1 展示了 SELECT 语句及其子句。

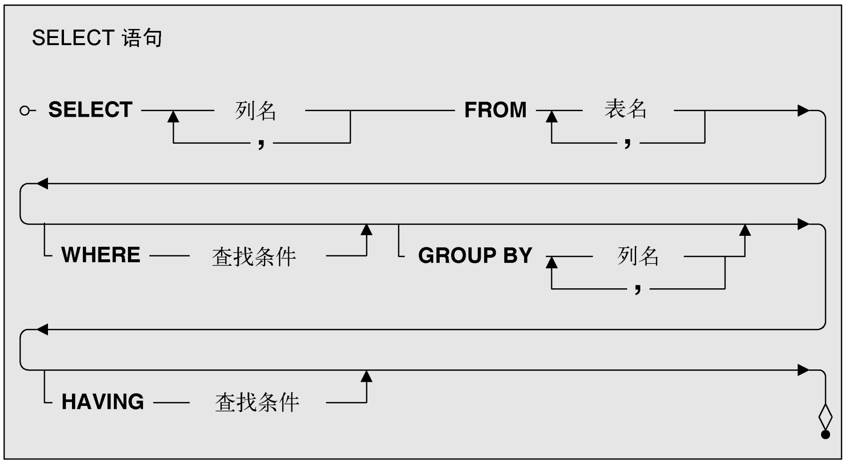


图 4-1 SELECT 语句的语法图

> ❖ **说明** 图 4-1 所示的语法图展示的是基本的 SELECT 语句。现在请暂时容忍我的这种做法，随着你使用 SQL 语句的经验逐渐丰富，我将不断更新和修改这个示意图，请耐心等待。

下面简单地总结了 SELECT 语句中的子句。

- SELECT：SELECT 语句的主子句，绝对必不可少。你使用它来指定要在查询结果集中包含哪些列，而这些列本身是从 FROM 子句中指定的表或视图中提取的（可同时从多个表中提取数据，这将在本书第三部分讨论）。在这个子句中，还可使用聚合函数（如 Sum(HoursWorked)）或数学表达式（如 Quantity * Price）。
- FROM：这是 SELECT 语句中第二重要的子句，也是必不可少的。你使用 FROM 子句来指定要从哪个表或视图中提取在 SELECT 子句中指定的列。你还可以以更复杂的方式使用这个子句，这将在本书后面讨论。
- WHERE：这个子句是可选的，用于筛选 FROM 子句返回的行。你在关键字 WHERE 后面指定一个表达式，严格地说这是一个**谓词**，其结果为真、假或未知。为测试这个表达式，可使用标准比较运算符、布尔运算符或特殊运算符。第 6 章将全面讨论 WHERE 子句。

- GROUP BY：在 SELECT 子句中使用聚合函数来生成汇总信息时，可使用 GROUP BY 子句将信息分组。数据库系统把关键字 GROUP BY 后面的指定列作为分组列。GROUP BY 子句是可选的，第 13 章将更深入地探讨。
- HAVING：HAVING 子句对分组的聚合函数结果进行筛选，它类似于 WHERE 子句，因为关键字 HAVING 后面也跟着一个结果为真、假或未知的表达式。为测试这个表达式，可使用标准比较运算符、布尔运算符或特殊运算符。HAVING 子句也是可选的，将在第 14 章更深入地探讨。

一开始你将使用非常简单的 SELECT 语句，因此这里重点介绍 SELECT 和 FROM 子句。在本书后面，随着创建的 SELECT 语句越来越复杂，我将逐一介绍其他子句。

4.3　说点题外话：数据和信息

对数据库进行查询前，有一点必须明白：**数据**和**信息**之间存在明显的差别。大体而言，数据是存储在数据库中的内容，而信息是从数据库检索得到的内容。你必须明白这种差别，因为这有助于看清本质。别忘了，设计数据库旨在向组织中的人员提供有意义的信息，但要提供这样的信息，必须满足两个条件：数据库包含合适的数据；数据库本身的结构为提供这种信息提供了支持。下面更深入地研究这些术语。

存储在数据库中的值是数据。数据是静态的，因为只要不以手动或自动化方式修改它，它的状态就不会变。图 4-2 显示了一些示例数据。

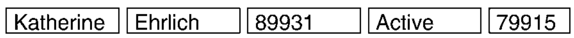


图 4-2　基本数据举例

从表面上看，这些数据毫无意义。例如，你难以确定 89931 表示的是什么。邮政编码？零件编号？即便知道它表示的是一位顾客的身份编号，但这个编号是 Katherine Ehrlich 的吗？在对这些数据进行处理前，无法确定这一点。对数据进行处理，使其变得有意义且有用后，数据就变成了信息。信息是动态的，因为不同于存储在数据库中的数据，信息是不断变化的，且可以以数不胜数的方式进行处理和呈现。你可以以 SELECT 语句结果的方式显示信息，在计算机屏幕上以表单的方式显示它，或作为报告打印到纸张上。这里要牢记的要点是，必须以特定的方式对数据进行处理，从而将其变成有意义的信息。

图 4-3 演示了如何将前一个示例中的数据转换为顾客屏幕上的信息。这表明通过特定的方式对数据进行操作，可让任何查看者都能看懂。

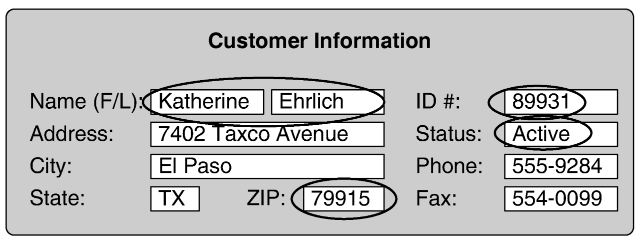


图 4-3　通过处理将数据变成信息

编写 SELECT 语句时，你使用子句来操作**数据**，而这个语句本身返回的是**信息**。明白了吗？

还有最后一点需要说说。当你执行 SELECT 语句时，它通常返回一行或多行信息，具体返回多少行取决于这条语句是如何编写的。这些行统称为**结果集**——本书余下的篇幅将始终使用这个术语。这个术语非常合理，因为使用关系型数据库时，处理的都是数据集（别忘了，从某种程度上说，关系模型是基于集合论的）。你可轻松地查看结果集中的信息，在很多情况下还可修改其中的数据。同样，这完全取决于 SELECT 语句是如何编写的。

我们言归正传，着手来使用 SELECT 语句。

4.4　将请求转换为 SQL

当你向数据库请求信息时，通常是以问题或隐含着问题的陈述表示的。例如，你可能像下面这样陈述请求。

“我们的顾客居住在哪些城市？”
“给我提供一个清单，其中包含所有员工及其电话号码。”
“当前我们提供了哪些类型的课程？”
“告诉我当前所有员工的姓名及其入职时间。”

知道要问什么后，就可将请求转换为更正规的语句。转换结果形如下面这样：

```
Select <item> from the <source>
```

首先研究你的请求，并将诸如“清单”“给我提供”“什么”“哪些”“谁”等词语替换为 Select。接下来，找出请求中的名词，并确定它表示的是一项数据还是数据项所在的表。如果是数据项，就将其替换为<item>；如果是表名，就将其替换为<source>。如果你对前面列出的第一个问题进行转换，结果将类似于下面这样：

```
Select city from the customers table
```

对请求进行转换后，还需根据 SQL 语法（如图 4-4 所示）将其转换为完整的 SELECT 语句。

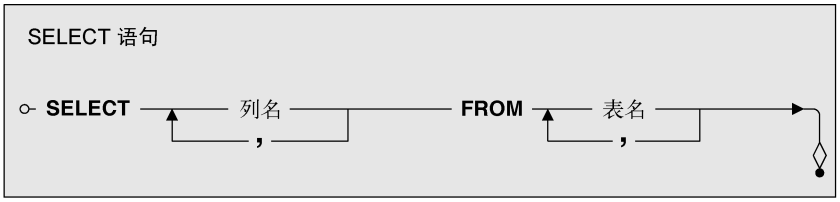


图 4-4　简单 SELECT 语句的语法

但在此之前，先得对转换结果进行整理。为此，可给如下内容添加删除线：不是列名的名词、不是表名的名词以及 SQL 语法中不使用的单词。下面演示了如何整理前面的转换结果：

`Select city from` ~~`the`~~ `customers` ~~`table`~~

将添加了删除线的单词删除后，便可得到一条完整的 SELECT 语句：

```
SELECT City FROM Customers
```

无论要向数据库发出什么请求，都可使用刚才介绍的三步法。事实上，在本书的大部分篇幅中，都使用了这种方法；建议你在刚开始学习如何编写 SELECT 语句时使用这种方法。然而，等你熟悉如何编写 SELECT 语句后，你将把这些步骤合并为一个无缝的操作。

别忘了，刚开始学习如何使用 SQL 时，处理的主要是列和表。图 4-4 所示的语法图反映了这一点，它在 SELECT 子句中使用了“列名”，在 FROM 子句中使用了“表名”。下一章将介绍如何在这些子句中使用其他术语来创建更复杂的 SELECT 语句。

你可能注意到了，前一个示例中的请求比较直观，很容易对其进行转换并找出转换结果中的列名。如果请求不那么直观，转换起来不容易，也难以确定在 SELECT 子句中该指定哪些列，该怎么办呢？最简单的办法是对请求进行改进，使其更为具体。例如，对于请求“告诉我有关顾客的信息”，可将其改写为“列出每个顾客的姓名、所在城市和电话号码”，这样请求将更为清晰而直观。如果改进请求不管用，还有另外两个办法。一是查看 SELECT 语句的 FROM 子句指定的表，看看其中是否包含有助于理清请求，从而能够更轻松地对请求进行转换的列名；二是更深入地研究请求，看看其中是否有隐含列名的词语。这两种办法是否管用取决于请求本身，你只需牢记，发现难以对请求进行转换时，还有其他办法可寻。下面通过示例来说明这两种方法的典型用法。

为演示第一种方法，假设你要对如下请求进行转换：

“我需要所有员工的姓名和地址。”

乍一看，这个请求好像很直观，但仔细看将发现一个小问题：虽然能够确定需要的表（Employees），但该请求没有提供任何相关的信息，无法帮助你确定应在 SELECT 子句中指定哪些列。虽然该请求中包含词语“姓名”和“地址”，但这些词语比较笼统。为解决这个问题，可查看根据请求确定的表，看看其中是否包含可用于替换这些词语的列。如果有，就在转换结果中使用这些列名（也可在转换结果中使用笼统的列名，如果这样做有助于更清晰地理解转换结果。但在 SQL 语法中，必须使用具体的列名）。这里，在 Employees 表（如图 4-5 所示）中查找可用来替换词语“姓名”和“地址”的列名。

EMPLOYEES	
EmployeeID	*PK*
EmpFirstName	
EmpLastName	
EmpStreetAddress	
EmpCity	
EmpState	
EmpZipCode	
EmpAreaCode	
EmpPhoneNumber	

图 4-5　Employees 表的结构

为满足词语“姓名”和“地址”提出的要求，需要使用这个表中的 6 个列：使用 EmpFirstName 和 EmpLastName 来替换请求中的词语“姓名”，并使用 EmpStreetAddress、EmpCity、EmpState 和 EmpZipCode 来替换词语“地址”。现在来完成前面重复了多次的整个请求转换过程（在转换结果中，使用笼统的列名，在 SQL 语句中使用具体的列名）。

“我需要所有员工的姓名和地址。”

转换	Select first name, last name, street address, city, state, and ZIP Code from the employees table（从员工表中选择名、姓、街道地址、城市、州和邮政编码）
整理	Select first name, last name, street address, city, state, ~~and~~ ZIP Code from ~~the~~ employees ~~table~~

SQL	`SELECT EmpFirstName, EmpLastName,`
	`   EmpStreetAddress, EmpCity,`
	`   EmpState, EmpZipCode`
	`FROM Employees`

❖ **说明**　这个示例清楚地演示了如何在 SELECT 子句中指定多列，本节后面将对此做更详细的讨论。

下面的示例演示了第二种方法，这要求在请求中查找隐含的列。假设你要对下面的请求进行转换：

“当前我们提供了哪些类型（kind）的课程？”

乍一看，好像很难对这个请求进行转换。这个请求没有指定任何列名，甚至没有指定要选择的数据项，因此无法进行完整的转换。面对这种情况，该怎么办呢？仔细查看请求中的每个词语，看看其中是否有隐含着 Classes 表中列名的词语。接着往下阅读前，请花点时间再研究一下这个请求。你找到了这样的词语吗？

在这里，词语“类型”可能隐含着 Classes 表中的一个列名。此话怎讲？因为课程类型也可视为课程类别（category）。如果 Classes 表中有表示类别的列，便有了完成转换，进而编写 SELECT 语句所需的列名。假设 Classes 表中有一个表示类别的列，因此可像下面这样对这个请求进行三步处理：

“当前我们提供了哪些类型的课程？”

转换	Select category from the classes table（从课程表中选择类别）
整理	Select category from ~~the~~ classes ~~table~~
SQL	`SELECT Category` `FROM Classes`

这个示例表明，这种方法要求使用同义词来替换请求中的某些词语。如果在请求中找到了可能隐含着列名的词语，可尝试用同义词来替换它，选择的同义词可能就是数据库中的一个列。如果最先想到的同义词不管用，可尝试另一个同义词，并不断重复这个过程，直到找到是列名的同义词，或发现原来的词语及其所有同义词都不是列名。

❖ **说明** 除非特别指出，否则示例中 SQL 语法部分的列名和表名都来自附录 B 中列出的示例数据库。本书后面所有的示例都如此。

4.4.1 更广阔的视野

在 SELECT 语句中可检索多列，这与检索单列一样容易。为此，只需在 SELECT 子句中列出要检索的列并用逗号分隔。在图 4-6 所示的语法图中，“列名”下方从右到左的箭头表示可指定多列，箭头线中间的逗号表示在 SELECT 子句中指定下一个列名前，必须先加上一个逗号。



图 4-6 在 SELECT 子句中指定多列的语法

通过在 SELECT 语句中指定多列，可回答类似于下面这样的问题：

“给我提供一个清单，其中包含所有的员工及其电话号码。”

转换	Select the last name, first name, and phone number of all our employees from the employees table（从员工表中选择所有员工的姓、名和电话号码）
整理	Select ~~the~~ last name, first name, ~~and~~ phone number ~~of all our employees~~ from ~~the~~ employees ~~table~~

SQL	`SELECT EmpLastName, EmpFirstName,`
	`   EmpPhoneNumber`
	`FROM Employees`

“我们出售哪些商品？这些商品的价格是多少？属于哪种类别？”

转换	Select the name, price, and category of every product from the products table（从商品表中选择所有商品的名称、价格和类别）
整理	Select ~~the~~ name, price, ~~and~~ category ~~of every product~~ from ~~the~~ products ~~table~~

SQL	`SELECT ProductName, RetailPrice,`
	`   Category`
	`FROM Products`

通过在 SELECT 语句中指定多列，可查看更多的信息。顺便说一句，在 SELECT 子句中指定列的顺序无关紧要，你想以什么样的顺序列出它们都可以。这提供了极大的灵活性，让你能够以众多方式查看同样的信息。

例如，假设有人要求你检索图 4-7 所示的表，他要向数据库发出的请求如下：

“给我提供一个科目（subject）清单，在其中依次列出每个科目的名称、类别和编码。”

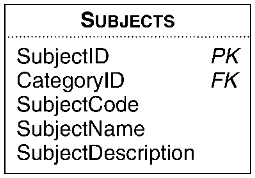


图 4-7　Subjects 表的结构

虽然请求者要求按特定的顺序列出指定的列，但你依然能够将这个请求转换为合适的 SELECT 语句。为此，只需在转换时按指定的顺序列出列名。下面演示了如何将这个请求转换为 SELECT 语句。

转换	Select the subject name, category ID, and subject code from the subjects table（从科目表中选择科目名、类别 ID 和科目代码）
整理	Select ~~the~~ subject name, category ID, ~~and~~ subject code from ~~the~~ subjects ~~table~~

SQL
```
SELECT SubjectName, CategoryID, SubjectCode
FROM Subjects
```

4.4.2　使用简写请求所有列

在 SELECT 子句中，可指定任意数量的列——实际上，可列出源表中所有的列。

在下面的 SELECT 语句中，指定了图 4-7 所示 Subjects 表中的所有列：

SQL
```
SELECT SubjectID, CategoryID, SubjectCode,
   SubjectName, SubjectDescription
FROM Subjects
```

指定源表中所有的列时，如果这个表包含的列很多，输入量将很大！所幸 SQL 标准规定了可使用简写方式——星号。通过使用这种简写，可极大地缩短语句。图 4-8 所示的语法图表明，在 SELECT 子句中，可使用星号来替代包含所有列的列表。

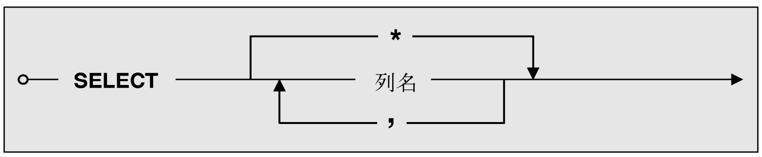


图 4-8　简写方式星号的使用语法

要指定 FROM 子句指定的表中所有的列，可在关键字 SELECT 后面加上星号。例如，使用这种简写方式时，前述 SELECT 语句将变成下面这样：

SQL
```
SELECT *
FROM Subjects
```

使用这条语句时，输入量显然更少！然而，以这样的方式编写 SELECT 语句时存在一个问题：星号表示源表**当前包含**的所有列，因此在源表中添加或删除列将影响 SELECT 语句返回的结果集（奇怪的是，SQL 标准竟然规定添加或删除列不应影响结果集）。仅当必须在结果集中始终看到相同的列时，这个问题才显得重要。如果在 SELECT 子句中使用星号，则有人删除了源表中的某些列时，数据库系统不会发出警告，但如果找不到你**显式**指定的列，数据库系统将发出警告。虽然就本书而言，这都不是问题，但当你开始 SQL 编程后，这个问题将变得很重要。我的经验规则是这样的：仅当需要快速创建查询以查看特定表中所有的信息时，才使用星号；在其他情况下，都应在查询中显式地指

定所需的列。这样查询将返回所需的信息（不多也不少），且更容易理解。

在前面所有的示例中，请求都很简单，只要求从一个表中检索列。本书第三部分将介绍如何处理更复杂的请求——它们要求从多个表中获取列。

4.5 消除重复行

使用SELECT语句时，肯定会遇到结果集中包含重复行的情形。如果你遇到这样的情形，不必恐慌，只需在SELECT语句中使用关键字 DISTINCT，就可消除所有的重复行。图 4-9 所示的语法图演示了关键字 DISTINCT 的用法。

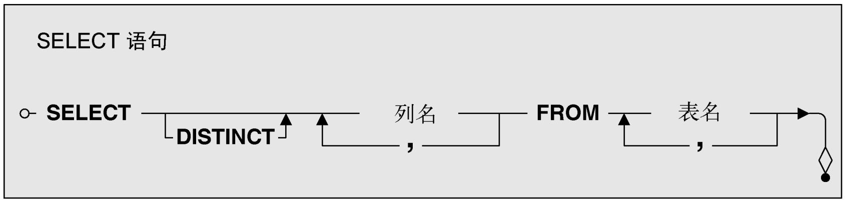


图 4-9 关键字 DISTINCT 的使用语法

从该图可知，DISTINCT 是一个可选的关键字，位于 SELECT 子句指定的列列表前面。关键字 DISTINCT 要求数据库系统逐行检查所有指定列的值，并将找到的重复行删除，再返回余下的独一无二的行。下面的示例表明，在合适的情况下，关键字 DISTINCT 可让结果大不相同。

假设要向数据库发出如下请求：

“哪些城市有保龄球联盟的成员？”

这个问题看似很容易回答，因此你像下面这样进行转换。

转换	Select city from the bowlers table（从投球手表中选择城市）
整理	Select city from ~~the~~ bowlers ~~table~~
SQL	`SELECT City` `FROM Bowlers`

这条 SELECT 语句存在的问题是，其结果集将列出 Bowlers 表中出现的每个城市名。例如，如果有 20 人来自贝尔维尤、7 人来自肯特、14 人来自西雅图，贝尔维尤将在结果集中出现 20 次，肯特将出现 7 次，西雅图将出现 14 次。显然，这里有冗余信息；你希望的结果是，在 Bowlers 表中出现的每个城市名都只在结果集中出现一次。要解决这个问题，可在 SELECT 语句中使用关键字 DISTINCT 来消除冗余信息。

下面来再次转换前述请求，但使用关键字 DISTINCT。注意，在“转换”和“整理”步骤中，都包含单词 distinct。

“哪些城市有保龄球联盟的成员？”

转换	Select the distinct city values from the bowlers table（从投球手表中选择不同的城市）
整理	Select ~~the~~ distinct city ~~values~~ from ~~the~~ bowlers ~~table~~
SQL	`SELECT DISTINCT City` `FROM Bowlers`

这条 SELECT 语句返回的结果集提供了所需的信息：在 Bowlers 表中出现的每个城市名都只出现一次。

检索多列时，也可使用关键字 DISTINCT。我们来修改前面的示例，从 Bowlers 表中同时获取城市和州。这种请求对应的 SELECT 语句类似于下面这样：

```
SELECT DISTINCT City, State FROM Bowlers
```

这条 SELECT 语句返回的结果集中不会包含相同的记录，并且还会区分属于不同州的同名城市。例如，它区分"Portland, ME"和"Portland, OR"，也区分"Hollywood, CA"和"Hollywood, FL"。需要指出的是，大多数数据库系统按指定的列对输出进行排序，因此前述值将按如下顺序排列："Hollywood, CA""Hollywood, FL""Portland, ME""Portland, OR"。然而，SQL 标准并没有要求必须按这样的顺序排列结果。如果要确保按特定的顺序排列结果，请阅读下一节，学习如何使用 ORDER BY 子句。

在使用得当的情况下，关键字 DISTINCT 是一个很有用的工具。仅当要在结果集中消除重复的行时，才使用这个关键字。

❖ **警告**　在包含图形用户界面的数据库系统中，通常可要求在可更新的行列网格中显示查询的结果集。在这种情况下，可在某行的某列中输入新值，而数据库系统将更新存储在表中的值（实际上，数据库系统在幕后替你执行了一个 UPDATE 查询，有关这方面的更详细信息，请参阅第 15 章）。

然而，在我研究过的所有数据库系统中，如果你使用了关键字 DISTINCT，结果集都是不可更新的。要更新某行中的某列，数据库系统必须能够唯一地标识它。当你使用关键字 DISTINCT 时，你看到的每行都可能是由数十个重复行合并而来的。如果你试图更新某列，数据库系统将无法确定该修改哪行。另外，数据库系统也不知道你是否要修改所有包含相同值的行。

4.6　对信息进行排序

本章开头说过，SELECT 操作可分成 3 个小操作：SELECT 语句、SELECT 表达式和 SELECT 查询。我还说过，可以以各种方式组合这些操作来完成复杂的请求。要对结果集中的行进行排序，也需要组合这些操作。

根据定义，SELECT 语句返回的结果集中的行是无序的：这些行的出现顺序通常取决于它们在表中的物理位置（实际的排列顺序通常是由数据库系统动态确定的：怎样能以效率最高的方式满足请求，就怎样排列）。要对结果集进行排序，唯一的方式是将 SELECT 语句嵌套在 SELECT 查询中，如图 4-10 所示。所谓 SELECT 查询，指的是包含 ORDER BY 子句的 SELECT 语句。SELECT 查询中的 ORDER BY 子句让你能够指定如何在结果集中排列各行。你在本书后面将学到，通过将 SELECT 语句嵌套在另一条 SELECT 语句或 SELECT 表达式中，可解决非常复杂的问题；然而，SELECT 查询不能嵌套。

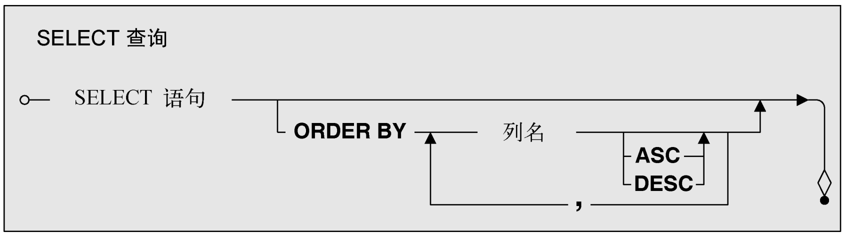


图 4-10　SELECT 查询的语法图

❖ **说明**　本书将始终使用 SQL 标准指定的术语或在大多数数据库系统中通用的术语。然而，在较早的 SQL 标准版本中，ORDER BY 子句被定义为**游标**（在应用程序中定义的一种对象）、**数组**（组成逻辑表的一系列值，如将在第 11 章讨论的子查询）或**标量子查询**（只返回一个值的查询）的一部分。有关游标和数组的完整讨论超出了本书的范围。

鉴于几乎在所有的 SQL 实现中，都可在 SELECT 语句末尾包含 ORDER BY 子句，并将其保存到视图中，我发明了术语 **SELECT 查询**来表示这样的语句。这让我能够讨论这样的概念，即对查询的最终输出进行排序，以便在

线显示或用在报告中。最新的 2016 标准使用术语**查询规范**（query specification）表示我所说的 **SELECT 语句**，并使用术语**查询表达式**（query expression）表示我所说的 **SELECT 查询**。对于这些结构，我将使用自己发明的术语，而不使用 SQL 标准定义的名称。

ORDER BY 子句让你能够按一列或多列对 SELECT 语句的结果集进行排序，还可指定按各列升序还是降序排列。在 ORDER BY 子句中，只能指定在 SELECT 子句中指定了的列（虽然 SQL 标准做了这样的规定，但有些厂商实现允许你完全无视这样的规定，在 ORDER BY 子句中指定 FROM 子句指定的表中的任何列）。在 ORDER BY 子句中指定多列时，必须用逗号将各列隔开。排序完毕后，SELECT 查询将立即返回最终的结果集。

❖ **说明**　ORDER BY 子句不会影响表中各行的物理顺序。如果要修改行的物理顺序，请参阅你使用的数据库软件的文档，了解完成这种任务的正确步骤。

4.6.1　重要的事先说：排序序列

介绍 SELECT 查询示例前，先来简单地说说排序序列（collating sequences）。ORDER BY 子句以什么样的方式对信息进行排序呢？这取决于数据库软件使用的排序序列。排序序列决定了操作系统指定的语言字符集中各个字符的优先顺序，例如，它指定了小写字母是否排在大写字母前面以及是否区分大小写。要确定数据库的默认排序序列，请参阅你使用的数据库软件的文档，或向数据库管理员咨询。有关排序序列的详细信息，请参阅 6.1.1 节。

4.6.2　排序

通过使用 ORDER BY 子句，可以以更有意义的方式呈现从数据库检索到的信息。这适用于简单请求，也适用于复杂请求。现在你可修改请求，在其中包含排序要求。例如，对于问题“我们当前提供哪些类别的课程”，可将其改为“按字母顺序列出我们提供的课程类别”。

着手编写 SELECT 查询前，必须调整对请求进行转换的方式，即在转换结果末尾添加一部分，阐述请求指定的排序要求。这样转换结果将形如下面这样：

```
Select <item> from the <source> and order by <column(s)>
```

请求包含诸如“按城市对结果排序”“按年份列出”或“按姓和名列出”等短语时，必须仔细研究请求，确定要指定用来排序的列。这项任务很简单，因为大多数人使用这样的措辞，且用于排序的列通常是不言自明的。确定要用来排序的列后，用它们替换转换结果中的<column(s)>即可。下面通过一个简单的请求来说明这个过程：

“按字母顺序列出我们提供的课程类别。”

转换	Select category from the classes table and order by category（从课程表中选择类别并按类别排序）
整理	Select category from ~~the~~ classes ~~table and~~ order by category

SQL	`SELECT Category`
	`FROM Classes`
	`ORDER BY Category`

在这个示例中，可假定用于排序的列为 Category，因为请求中只提到了这个列。还可假定按升序排列，因为请求没有明确地要求按降序排列。这些假设是可行的。SQL 标准指出，如果没有指定排列顺序，默认自动按升序排列；但如果要明确地指定按升序排列，可在 ORDER BY 子句中的 Category 后面加上 ASC。

在下面的请求中，更明确地指出了用于排序的列：

“按邮政编码顺序列出供应商。”

转换	Select vendor name and ZIP Code from the vendors table and order by ZIP Code（从供应商表中选择供应商名称和邮政编码，并按邮政编码排序）
整理	Select vendor name ~~and~~ ZIP Code from ~~the~~ vendors ~~table and~~ order by ZIP Code

SQL

```
SELECT VendName, VendZipCode
FROM Vendors
ORDER BY VendZipCode
```

对大多数人来说，如果他想按降序查看信息，通常会指出这一点。要按降序显示结果集时，可在 ORDER BY 子句中相应的列后面加上关键字 DESC。例如，下面演示了如何修改前一个示例中的 SELECT 语句，以便按邮政编码降序排列信息。

SQL

```
SELECT VendName, VendZipCode
FROM Vendors
ORDER BY VendZipCode DESC
```

> ❖ **说明**　如果有多个供应商的邮政编码相同，而你又没有在 ORDER BY 子句中指定如何排列这些供应商，数据库系统将决定如何排列它们。

接下来的示例演示了一个更复杂的请求，它要求根据多列进行排序。与前两个示例相比，这个示例唯一不同的地方是，它要求在 ORDER BY 子句中指定多个列。请注意，从图 4-10 所示的语法图可知，必须用逗号将各列分开。

“显示员工的姓名、电话号码和 ID 号，并按姓和名排列。”

转换	Select last name, first name, phone number, and employee ID from the employees table and order by last name and first name（从员工表中选择姓、名、电话号码和员工 ID，并按姓和名排序）
整理	Select last name, first name, phone number, ~~and~~ employee ID from ~~the~~ employees ~~table and~~ order by last name ~~and~~ first name

SQL

```
SELECT EmpLastName, EmpFirstName,
   EmpPhoneNumber, EmployeeID
FROM Employees
ORDER BY EmpLastName, EmpFirstName
```

对于在 ORDER BY 子句中指定的列，可做的一件有趣的事情是，给每列指定不同的排列顺序。例如，在前一个示例中，可将包含姓的那列的排列顺序指定为降序，而将包含名的那列的排列顺序指定为升序。这样修改后的 SELECT 语句类似于下面这样：

SQL

```
SELECT EmpLastName, EmpFirstName,
   EmpPhoneNumber, EmployeeID
FROM Employees
ORDER BY EmpLastName DESC, EmpFirstName ASC
```

按升序排列时，可不指定关键字 ASC，但指定它可让语句更易于理解。

前一个示例提出了一个有趣的问题：在 ORDER BY 子句中，指定各列的顺序重要吗？答案是肯定的！这种顺序很重要，因为数据库系统将以从左到右的顺序评估 ORDER BY 子句中的各列。另外，指定的列越多，这种顺序就越重要。在 ORDER BY 子句中，务必按正确的顺序指定各列，以确保以合适的顺序排列结果。

> ❖ **说明**　微软的数据库产品（Microsoft Office Access 和 Microsoft SQL Server）包含一个有趣的扩展，让你能够在 SELECT 子句中使用关键字 TOP 请求部分行（具体是哪些行取决于 ORDER BY 子句）。例如，要找出数据库 Sales Orders 中价格最高的 5 种商品，可像下面这样做：

```
SELECT TOP 5 ProductName, RetailPrice
   FROM Products
   ORDER BY RetailPrice DESC
```

这个查询根据价格以降序方式将Products表中的所有行排序，并返回最前面的5行。在这两个数据库系统中，还可以以百分比的方式指定要返回的行数。例如，要找出价格排名在前10%的商品，可像下面这样做：

```
SELECT TOP 10 PERCENT ProductName, RetailPrice
   FROM Products
   ORDER BY RetailPrice DESC
```

实际上，如果你在视图中指定ORDER BY，SQL Server要求必须包含关键字TOP。因此，如果要返回所有的行，必须指定TOP 100 PERCENT。有鉴于此，SQL Server中所有包含ORDER BY子句的视图都指定了TOP 100 PERCENT。Microsoft Access没有这样的限制。

4.7 保存所做的工作

所有主流数据库软件程序都提供了保存SELECT语句的途径。保存语句后，需要向数据库发出同样的请求时，就无须重新创建它们。保存SELECT语句时，请指定有意义的名称，以帮助你记住它们提供的是什么样的信息。另外，如果数据库软件允许，请编写简短的描述，对语句的用途进行说明。在没有看到SELECT语句，而又想知道当初为何创建它时，描述的价值将显现出来。

对于保存的SELECT语句，有些数据库程序将其归类为查询，还有一些数据库程序将其归类为视图、函数或存储过程。无论将保存的语句称为什么，每个数据库程序都提供了相应的途径，让你能够执行（运行）它们并使用返回的结果集。

❖ **说明** 在本节余下的篇幅中，我们将使用术语**查询**来表示保存的SELECT语句，并使用术语**执行**来表示使用查询。

执行查询的常用方法有两种。一是交互式工具（如工具栏或查询网格中的命令按钮），二是代码块。你将经常使用第一种方法；至于第二种方法，在你着手使用数据库软件的编程语言前，根本不用考虑。虽然教你如何创建和使用SQL语句是我的职责，但学习如何在数据库软件程序中创建、保存和执行它们却是你的职责。

❖ **说明** 所有示例语句和问题都保存在了用于Microsoft Access、Microsoft SQL Server、MySQL和PostgreSQL的示例数据库中。在Microsoft Access中，这些保存的语句被称为查询，而在其他数据库系统中被称为视图。

不同于其他大多数数据库系统，在视图中使用ORDER BY方面，SQL Server有个小怪癖。虽然你可将包含ORDER BY子句的查询保存为视图，但如果你从程序、其他视图或命令行调用这种视图，SQL Server将忽略其中的ORDER BY子句。因此，在SQL Server中保存的包含ORDER BY的查询不会以指定的顺序返回结果。要按指定的顺序返回结果，要么在设计工具中打开并执行查询，要么在调用查询时添加相应的ORDER BY子句，如SELECT * FROM <视图名> ORDER BY <原始排序规范>。

4.8 语句举例

介绍了SELECT语句和SELECT查询的基本特征后，我们来看一些示例，它们演示了如何在各种情形下使用这些操作。这些示例都指出了涉及的数据库，还演示了如何使用SELECT语句和SELECT查询以及确定要检索哪些列的两种补充方法。另外，在SQL语法行后面，还列出了这些操作返回的示例结果集。结果集前面的名称有两个作用：既标识了结果集本身，也是我给示例中的SQL语句指定的名称。

你可能会问：为何要给每条 SQL 语句指定名称呢？因为我保存了它们！实际上，无论是这里还是本书后面的示例，对于其中出现的每条 SQL 语句，我都将其保存并命名了。每条 SQL 语句都存储在相应的示例数据库（示例指出了是哪个）中，同时对于本章的示例查询，我还给其名称加上了前缀 CH04。可按本书前言中的说明在你的计算机中加载这些示例，这样你将有机会在自己动手编写 SQL 语句前看到它们是如何执行的。

❖ **说明**　提醒一下，这些示例涉及的所有列名和表名都来自示例数据库，有关这些示例数据库的结构，请参阅附录 B。另外别忘了，对于你运行的查询，如果其中不包含 ORDER BY 子句，返回的行的排列顺序是不确定的。在大多数情况下，无论在哪种示例数据库（Microsoft SQL Server、Microsoft Office Access、MySQL 和 PostgreSQL）中，这种查询返回的行的排列顺序都不一定与本书列出的顺序相同。如果你在其他数据库系统中使用 SQL 脚本加载这些示例，将发现返回的行数以及这些行包含的内容都与这里列出的相同，但排列顺序可能不同。

- **Sales Orders 数据库**

“显示所有供应商的名称。”

转换	Select the vendor name from the vendors table（从供应商表中选择供应商名称）
整理	Select ~~the~~ vendor name from ~~the~~ vendors ~~table~~
SQL	`SELECT VendName` `FROM Vendors`

CH04_Vendor_Names（10 行）

VendName
Shinoman, Incorporated
Viscount
Nikoma of America
ProFormance
Kona, Incorporated
Big Sky Mountain Bikes
Dog Ear
Sun Sports Suppliers
Lone Star Bike Supply
Armadillo Brand

“我们销售哪些商品？它们的价格是多少？”

转换	Select product name, retail price from the products table（从商品表中选择商品名称和零售价）
整理	Select product name, retail price from ~~the~~ products ~~table~~
SQL	`SELECT ProductName, RetailPrice` `FROM Products`

CH04_Product_Price_List（40 行）

ProductName	Retail Price
Trek 9000 Mountain Bike	$1,200.00
Eagle FS-3 Mountain Bike	$1,800.00
Dog Ear Cyclecomputer	$75.00
Victoria Pro All Weather Tires	$54.95

（续）

ProductName	Retail Price
Dog Ear Helmet Mount Mirrors	$7.45
Viscount Mountain Bike	$635.00
Viscount C-500 Wireless Bike Computer	$49.00
Kryptonite Advanced 2000 U-Lock	$50.00
Nikoma Lok-Tight U-Lock	$33.00
Viscount Microshell Helmet	$36.00
<< 其他行 >>	

“哪些州有我们的顾客？”

转换	Select the distinct state values from the customers table（从顾客表中选择不同的州）
整理	Select ~~the~~ distinct state ~~values~~ from ~~the~~ customers ~~table~~
SQL	`SELECT DISTINCT CustState` `FROM Customers`

CH04_Customer_States（4 行）

CustState
CA
OR
TX
WA

- **Entertainment Agency 数据库**

“列出所有的演唱组合及其所在的城市，并按城市和艺名升序排列结果。”

转换	Select city and stage name from the entertainers table and order by city and stage name（从演唱组合表中选择城市和艺名并按城市和艺名排序）
整理	Select city ~~and~~ stage name from ~~the~~ entertainers ~~table and~~ order by city ~~and~~ stage name

SQL	`SELECT EntCity, EntStageName`
	`FROM Entertainers`
	`ORDER BY EntCity ASC, EntStageName ASC`

CH04_ Entertainer_Locations（13 行）

EntCity	EntStageName
Auburn	Caroline Coie Cuartet
Auburn	Topazz
Bellevue	Jazz Persuasion
Bellevue	Jim Glynn
Bellevue	Susan McLain
Redmond	Carol Peacock Trio
Redmond	JV & the Deep Six
Seattle	Coldwater Cattle Company
Seattle	Country Feeling
Seattle	Julia Schnebly
<< 其他行 >>	

“给我一个清单，列出有演出的日期。至于各个日期有多少场演出，我不关心。”

转换	Select the distinct start date values from the engagements table（从演出合约表中选择不同的开始日期）
整理	Select ~~the~~ distinct start date ~~values~~ from ~~the~~ engagements ~~table~~
SQL	`SELECT DISTINCT StartDate` `FROM Engagements`

CH04_ Engagement_Dates（64 行）

StartDate
2017-09-02
2017-09-11
2017-09-12
2017-09-16
2017-09-18
2017-09-19
2017-09-25
2017-09-30
2017-10-01
2017-10-02
<< 其他行 >>

- **School Scheduling 数据库**

“我能查看完整的课程信息吗？”

转换	Select all columns from the classes table（选择课程表中的所有列）
整理	Select ~~all columns~~ * from ~~the~~ classes ~~table~~
SQL	`SELECT *` `FROM Classes`

CH04_ Class_Information（132 行）

ClassID	SubjectID	ClassRoomID	Credits	StartDate	StartTime	Duration	其他列
1000	11	1231	5	2017-09-12	10:00	50	…
1002	12	1619	4	2017-09-11	15:30	110	…
1004	13	1627	4	2017-09-11	08:00	50	…
1006	13	1627	4	2017-09-11	09:00	110	…
1012	14	1627	4	2017-09-12	13:00	170	…
1020	15	3404	4	2017-09-12	13:00	110	…
1030	16	1231	5	2017-09-11	11:00	50	…
1031	16	1231	5	2017-09-11	14:00	50	…
1156	37	3443	5	2017-09-11	16:00	50	…
1162	37	3443	5	2017-09-11	09:00	80	…
<< 其他行 >>							

“给我一个清单，列出校园内所有的大楼及其层数，并按大楼名升序排列。”

转换	Select building name and number of floors from the buildings table, ordered by building name（从大楼表中选择大楼名称和层数并按大楼名称排序）
整理	Select building name ~~and~~ number of floors from ~~the~~ buildings ~~table~~, order~~ed~~ by building name

SQL	`SELECT BuildingName, NumberOfFloors`
	`FROM Buildings`
	`ORDER BY BuildingName ASC`

CH04_ Building_List（6 行）

BuildingName	NumberOfFloors
Arts and Sciences	3
College Center	3
Instructional Building	3
Library	2
PE and Wellness	1
Technology Building	2

- **Bowling League 数据库**

“联赛在哪些地方举行？”

转换	Select the distinct tourney location values from the tournaments table（从联赛表中选择不同的比赛地点）
整理	Select ~~the~~ distinct tourney location ~~values~~ from ~~the~~ tournaments ~~table~~
SQL	`SELECT DISTINCT TourneyLocation` `FROM Tournaments`

CH04_ Tourney_Locations（7 行）

TourneyLocation
Acapulco Lanes
Bolero Lanes
Imperial Lanes
Red Rooster Lanes
Sports World Lanes
Thunderbird Lanes
Totem Lanes

“给我一个清单，列出所有比赛日期和比赛地点，并按日期降序排列、按比赛地点升序排列。”

转换	Select tourney date and location from the tournaments table and order by tourney date in descending order and location in ascending order（从联赛表中选择比赛日期和比赛地点，并按日期降序排列、按比赛地点升序排列）
整理	Select tourney date ~~and~~ location from ~~the~~ tournaments ~~table and~~ order by tourney date ~~in~~ desc~~ending order and~~ location ~~in~~ asc~~ending order~~

SQL	`SELECT TourneyDate, TourneyLocation`
	`FROM Tournaments`
	`ORDER BY TourneyDate DESC, TourneyLocation ASC`

CH04_ Tourney_Dates（20 行）

TourneyDate	TourneyLocation
2018-08-16	Totem Lanes
2018-08-09	Imperial Lanes
2018-08-02	Sports World Lanes
2018-07-26	Bolero Lanes
2018-07-19	Thunderbird Lanes
2018-07-12	Red Rooster Lanes
2017-12-04	Acapulco Lanes
2017-11-27	Totem Lanes
2017-11-20	Sports World Lanes
2017-11-13	Imperial Lanes
<< 其他行 >>	

- **Recipes 数据库**

“我们提供哪些类型的菜品？每种类型包含的菜品都叫什么？你能按类型和菜品名排列这些信息吗？”

转换	Select recipe class ID and recipe title from the recipes table and order by recipe class ID and recipe title（从菜品表中选择菜品类型 ID 和菜品名，并按类型 ID 和菜品名排列）
整理	Select recipe class ID ~~and~~ recipe title from ~~the~~ recipes ~~table and~~ order by recipe class ID ~~and~~ recipe title

SQL	`SELECT RecipeClassID, RecipeTitle`
	`FROM Recipes`
	`ORDER BY RecipeClassID ASC, RecipeTitle ASC`

CH04_ Recipe_Classes_And_Titles（15 行）

RecipeClassID	RecipeTitle
1	Fettuccini Alfredo
1	Huachinango Veracruzana (Red Snapper, Veracruz style)
1	Irish Stew
1	Pollo Picoso
1	Roast Beef
1	Salmon Filets in Parchment Paper
1	Tourtière (French-Canadian Pork Pie)
2	Asparagus
2	Garlic Green Beans
3	Yorkshire Pudding
	<< 其他行 >>

“给我一个清单，列出菜品表包含哪些菜品类型 ID。”

转换	Select the distinct recipe class ID values from the recipes table（从菜品表中选择不同的菜品类型 ID）
整理	Select ~~the~~ distinct recipe class ID ~~values~~ from ~~the~~ recipes ~~table~~
SQL	`SELECT DISTINCT RecipeClassID` `FROM Recipes`

CH04_Recipe_Class_Ids（6 行）

RecipeClassID
1
2
3
4
5
6

4.9　小结

本章介绍了 SELECT 操作，它是 SQL 中的 4 种数据操作之一（其他 3 种数据操作为 UPDATE、INSERT 和 DELETE，将在第五部分讨论）。SELECT 操作可分为 3 种小操作：SELECT 语句、SELECT 表达式和 SELECT 查询。

接下来讨论了 SELECT 语句及其包含的子句。就从数据库中检索信息而言，SELECT 和 FROM 子句是必不可少的基本子句，而其他子句（WHERE、GROUP BY 和 HAVING）用于对 SELECT 子句返回的信息执行有条件的处理和筛选。

然后讨论了数据和信息的不同之处。存储在数据库中的值是数据，而信息是经特定方式处理，让查看者能够看明白的数据。SELECT 语句返回的成行信息称为结果集。

接着讨论了信息检索。首先介绍的是 SELECT 语句的基本形式，以及如何使用三步法将请求转换为合适的 SQL 语法，从而创建合适的 SELECT 语句。你还了解到可在 SELECT 子句中指定多列，从而从数据库检索更多的信息。然后简要地介绍了关键字 DISTINCT，这是一种消除结果集中重复行的方法。

然后介绍了 SELECT 查询，以及如何结合使用 SELECT 查询和 SELECT 语句来对 SELECT 语句的结果集进行排序。这是必需的，因为 SELECT 查询是唯一一种包含 ORDER BY 子句的 SELECT 操作。ORDER BY 子句用于根据一列或多列对信息进行排序，而每列都可以有自己的排序规范（升序或降序）。然后，简要地讨论了如何保存 SELECT 语句，你了解到可将语句保存为查询或视图，供以后使用。

最后，本章列举了很多示例，它们使用了示例数据库中的各种表。这些示例演示了如何在典型场景下使用本章介绍的各种概念和方法。下一章将深入介绍 SELECT 子句，并演示如何检索除简单列外的其他信息。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

4.10　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**
  (1) “显示所有员工的完整信息。”
  解决方案见 CH04_Employee_Information（8 行）。
  (2) “给我一个按字母顺序排列的清单，其中包含每个供应商的名称及其所在的城市。”
  解决方案见 CH04_Vendor_Locations（10 行）。
- **Entertainment Agency 数据库**
  (1) “给我所有经纪人的姓名和电话号码，并根据姓/名排列。”
  解决方案见 CH04_Agent_Phone_List（9 行）。
  (2) “给我有关所有演出合约的信息。”
  解决方案见 CH04_Engagement_Information（111 行）。

(3) “列出所有演出合约及其开始日期，根据开始日期降序排列，并根据演出合约升序排列。”
解决方案见 CH04_Scheduled_Engagements（111 行）。

- **School Scheduling 数据库**

(1) “给我一个清单，列出学校提供的所有科目。”
解决方案见 CH04_Subject_List（56 行）。

(2) “我们的教员都是什么职称（title）？”
解决方案见 CH04_Faculty_Titles（3 行）。

(3) “列出所有教员的姓名和电话号码，并根据姓和名排列。”
解决方案见 CH04_Staff_Phone_List（27 行）。

- **Bowling League 数据库**

(1) “按字母顺序列出所有球队。”
解决方案见 CH04_Team_List（10 行）。

(2) “列出每个投球手的所有得分信息。”
解决方案见 CH04_Bowler_Score_Information（1344 行）。

(3) “按字母顺序列出所有的投球手及其地址。”
解决方案见 CH04_Bowler_Names_Addresses（32 行）。

- **Recipes 数据库**

(1) “列出当前记录的所有食材。”
解决方案见 CH04_Complete_Ingredient_List（79 行）。

(2) “列出所有菜品的信息，并根据菜品名按字母顺序排列。”
解决方案见 CH04_Main_Recipe_Information（15 行）。

第 5 章 获取除简单列外的其他信息

“事实是不容改变的。”

——Tobias Smollett，*Gil Blas De Santillane*

本章涵盖如下主题：

- 何谓表达式
- 你要表示哪些类型的数据
- 修改数据类型：CAST 函数
- 指定显式值
- 表达式类型
- 在 SELECT 子句中使用表达式
- 空值：Null
- 语句举例
- 小结
- 练习

第 4 章介绍了如何使用 SELECT 语句从表中的一列或多列检索信息。需要向数据库发出简单请求以获悉一些基本事实时，这种方法很有用，但要处理复杂的请求，必须具备更丰富的 SQL 知识。本章首先介绍表达式，它让你能够操纵表中数据，以计算或生成新的信息。接下来讨论列中存储的数据类型，这对你创建的查询和表达式有重要影响。此外还将简要地讨论 CAST 函数，这个函数可用来修改表达式中包含的数据的类型。你将学习如何创建可在查询中创造性使用的常量（字面量），学习如何使用字面量和列值来创建表达式，以及如何使用表达式来操纵从中派生信息的数据，从而调整使用 SELECT 语句检索得到信息的范围。最后将探索特殊值 Null，以及它将对你使用表列编写表达式带来什么影响。

5.1 何谓表达式

要获取除简单列外的其他信息，需要创建表达式。**表达式**是一种涉及数字、字符串或日期和时间的运算，可使用表列中的值、常量（字面量）或两者的组合。有关如何生成字面量，将在本章后面介绍。数据库执行完表达式指定的运算后将返回一个值，供 SQL 语句做进一步处理。可使用表达式来增大或缩小从数据库检索的信息的范围。回答类似于“如果……将如何”这样的问题时，表达式很有用。下面是一些可使用表达式来回答的问题类型。

“订购的每种商品的总价是多少？”
“给我一个员工邮寄清单，将姓放在最前面。”
“显示每个课程的开始时间、结束时间和时长。”
“显示每位投球手的让分得分（handicap score）和原始得分的差。”
“每个演出合约的小时收费是多少？”
“如果将商品的价格提高 5%，结果将如何？”

你将在本章学到，学习表达式将极大地提高你的 SQL 技能。通过在查询中使用表达式，可对列中的原始数据进行整合，生成更有意义的结果。等你阅读第 6 章时，也将发现表达式很有用；你可使用表达式来筛选数据以及整合来自相关表中的数据。

5.2　你要表示哪些类型的数据

在表达式中使用的数据类型会影响表达式返回的值，因此我们先来看看 SQL 标准提供的一些数据类型。对于数据库中的每列，都指定了**数据类型**，这决定了它能存储什么样的值。数据类型还决定了可对列的值执行什么样的运算。要创建字面量值，或者在表达式中结合使用列和字面量，让表达式有意义并返回合适的值，你必须了解基本的数据类型。

SQL 标准定义了 7 大类数据类型：字符、国家字符（national character）、二进制、数值、布尔值、日期时间和间隔，其中每个类别都包含一种或多种名称相同的数据类型。下面简要地介绍一下这些类别及其包含的数据类型（在下面的项目列表中，我将数值分成了两小类：精确数值和近似数值）。

- **字符**：字符数据类型存储由一个或多个可打印字符组成的定长或变长的字符串。在这种数据类型中，通常可存储 ASCII（美国信息交换标准码）或 EBCDIC（广义二进制编码的十进制交换码）字符集中的字符。定长字符数据类型名为 CHARACTER 或 CHAR，而变长字符数据类型名为 CHARACTER VARYING、CHAR VARYING 或 VARCHAR。对于字符数据类型，可指定要存储的数据长度，但可指定的最大长度是由数据库系统规定的（这条规则也适用于国家字符数据类型）。如果字符串的长度超过了系统规定的最大长度（通常为 255 或 1024 个字符），就必须使用数据类型 CHARACTER LARGE OBJECT、CHAR LARGE OBJECT 或 CLOB。在很多系统中，CLOB 的别名为 TEXT 或 MEMO。
- **国家字符**：国家字符数据类型与字符数据类型相同，但可存储的字符来自 ISO 定义的外国语言字符集。定长的国家字符数据类型名为 NATIONAL CHARACTER、NATIONAL CHAR 或 NCHAR，而变长的国家字符数据类型名为 NATIONAL CHARACTER VARYING、NATIONAL CHAR VARYING 或 NCHAR VARYING。字符串长度超过系统规定的最大长度（通常为 255 或 1024 个字符）时，必须使用数据类型 NATIONAL CHARACTER LARGE OBJECT、NCHAR LARGE OBJECT 或 NCLOB。在很多系统中，NCLOB 的别名为 NTEXT。
- **二进制**：对于图像、声音、视频或复杂的嵌入文档（如字处理文件或电子表格）等二进制数据，使用数据类型 BINARY LARGE OBJECT（BLOB）来存储。在很多系统中，这种数据类型名为 BINARY、BIT 或 BIT VARYING。
- **精确数值**：这种数据类型存储整数和小数。精确数值的精度（有效位数）和范围（小数点右边的位数）可由用户定义，但不能超过数据库系统的限制。这种数据类型名为 NUMERIC、DECIMAL、DEC、SMALLINT、INTEGER、INT 或 BIGINT。必须牢记的一点是，在 SQL 标准和大多数数据库系统中，BIGINT 的取值范围比 INTEGER 大，而 INTEGER 的取值范围又比 SMALLINT 大。有关这些数据类型的具体取值范围，请参阅数据库系统文档。有些数据库系统还支持数据类型 TINYINT，其取值范围比 SMALLINT 还小。
- **近似数值**：这种数据类型存储小数和指数，名为 FLOAT、REAL 或 DOUBLE PRECISION。近似数值数据类型本身没有精度和范围，但 SQL 标准允许用户给 FLOAT 指定精度。这些数据类型的范围都是由数据库系统指定的。请注意，在 SQL 标准和大多数数据库系统中，数据类型 DOUBLE PRECISION 的取值范围比 REAL 和 FLOAT 大。有关这些数据类型的具体取值范围，请参阅数据库系统文档。
- **布尔值**：这种数据类型存储真假值，通常只使用一个二进制位。有些系统使用 BIT、INT 或 TINYINT 来存储这种数据类型。
- **日期时间**：这种数据类型存储日期、时间及其组合。SQL 标准规定日期格式为年-月-日，而时间值是基于 24 小时制的。虽然在大多数数据库系统中，可使用更常见的日期格式月/日/年或日/月/年以及基于 A.M./P.M. 的时间值，但贯穿本书都将使用 SQL 标准规定的日期和时间格式。这类数据类型有 3 种——DATE、TIME 和 TIMESTAMP，其中 TIMESTAMP 可用于存储日期和时间组合。请注意，这些数据类型的名称和用法因数据库系统而异，有些数据库系统使用数据类型 DATE 来同时存储日期和时间，而有些使用数据类型 TIMESTAMP 或 DATETIME，详情请参阅你使用的数据库系统的文档。

- ❑ **间隔**：这种数据类型存储两个日期时间值相隔的时间量，表示为年/月/天/时间或天/时间。并非所有的主流数据库系统都支持数据类型 INTERVAL，详情请参阅你使用的数据库系统的文档。

除 SQL 标准指定的数据类型（前面的数据类型类别列表中列出了其中的一些）外，很多数据库系统都提供了其他数据类型，这些数据类型被称为**扩展数据类型**。扩展数据类型的例子包括 MONEY/CURRENCY 和 SERIAL/ROWID/AUTOINCREMENT/IDENTITY（用作独一无二的行标识符）。

鉴于我们的重点是 SQL 的**数据操作**部分，你只需关注你使用的数据库系统支持的数据类型的取值范围。这种知识有助于确保你定义的表达式得以正确地执行，因此务必熟悉你使用的 RDBMS 程序提供的数据类型。

5.3　修改数据类型：CAST 函数

创建表达式时必须小心，要确保列和字面量的数据类型与要执行的操作兼容。例如，将字符数据与数字相加毫无意义，但如果字符列或字面量包含的是数字，可使用 CAST 函数进行转换，再将其与另一个数字相加。图 5-1 演示了 CAST 函数，几乎所有商用数据库系统都支持它。

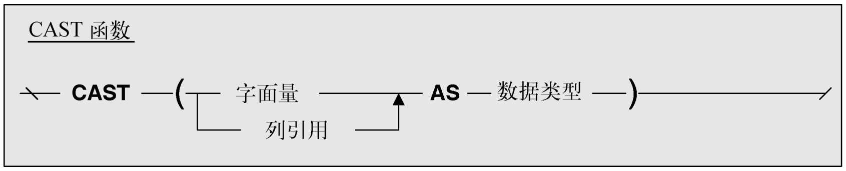


图 5-1　CAST 函数的语法图

CAST 函数将字面量或列值转换为特定的数据类型，帮助确保表达式中的值是**兼容**的。所谓兼容，指的是表达式中的所有列或字面量都是字符、数字或日期时间（与其他规则一样，这个规则也有例外，这将在后面介绍）。通常，要让表达式定义的操作能够正确执行，其中所有的值都必须是兼容的，否则数据库系统可能显示错误消息。

> ❖ **说明**　虽然大多数商用数据库系统支持 CAST 函数，但确实有不支持的。不支持 CAST 函数的数据库系统提供了一组自定义函数，可实现同样的效果，详情请参阅数据库系统文档。

将列值或字面量从一种数据类型转换为另一种数据类型是一项简单而直观的任务，但必须牢记如下限制。

- ❑ 不要用 5 磅的盒子装 10 磅的东西。前面说过，对于字符数据类型，可指定要在其中最多存储多长的数据。尝试将一种字符类型（如 VARCHAR）转换为另一种字符类型（如 CHARACTER）时，如果原始列或字面量中的数据长度超过了目标数据类型指定的最大长度，数据库系统将截断原始字符串。在截断前，数据库系统将发出警告，指出这一点。
- ❑ 不要以圆纳方。可将字符列或字符字面量转换为其他任何数据类型，但它必须能够转换为目标数据类型。例如，可将 5 字符的邮政编码转换为数字，但如果 ZIP Code 列包含的是带字母的加拿大或欧洲邮政编码，将发生错误。请注意，将字符列转换为数值或日期时间时，数据库系统将忽略开头和末尾的所有空格。另外，大多数商用数据库系统支持各种可解读为日期或时间的字符串。有关这方面的详情，请参阅数据库系统文档。
- ❑ 不要用 5 磅的盒子装 10 磅的东西（第 2 版）。将数值列转换为另一种数值类型时，其当前值必须在目标数据类型的取值范围内。例如，如果将大于 32767 的 REAL 值转换为 SMALLINT，很可能会出错。另外，将带小数的数字转换为 INTEGER 或 SMALLINT，将删除小数部分或做四舍五入处理，具体如何处理取决于数据库系统。
- ❑ 在一定条件下可以圆纳方。将数值列转换为字符数据类型时，可能出现如下三种情况。
  (1) 转换成功。
  (2) 如果它比字符列的指定长度短，数据库系统将用空格填充。
  (3) 如果数值的字符表示长于字符列的指定长度，数据库系统将引发错误。

❖ **说明**　虽然在将值从一种数据类型转换为另一种数据类型方面，SQL 标准做了这些限制，但你使用的数据库系统所做的限制可能更宽松。有些数据库系统提供了自动转换功能，而不要求你使用 CAST 函数。例如，在有些数据库系统中，可将数字与文本拼接，还可将包含数字的文本与其他数字相加，而无须显式地进行转换。详情请参阅你使用的数据库系统的文档。

必须指出的是，前面并没有列出 SQL 标准所做的所有限制，而只列出了与本书使用的数据类型相关的限制。有关数据类型和数据转换问题的更深入讨论，请参阅附录 D 列出的任何图书。

阅读本书后续内容时，请务必牢记 CAST 函数。在情况合适时，我都会使用它来确保数据类型是兼容的。

5.4　指定显式值

SQL 标准规定，可在 SELECT 语句中使用表达式来指定常量值，如字符串、数字、日期、时间以及这些值的组合，这让你能够改进 SELECT 语句返回的信息。SQL 标准将这些值归类为**字面量**，并指定了定义它们的方式。

5.4.1　字符串字面量

字符串字面量是用单引号括起的字符序列。我知道，你以前可能使用过双引号来将字符串括起，但这里说的是 SQL 标准定义的字符串。图 5-2 是字符串字面量的语法图。

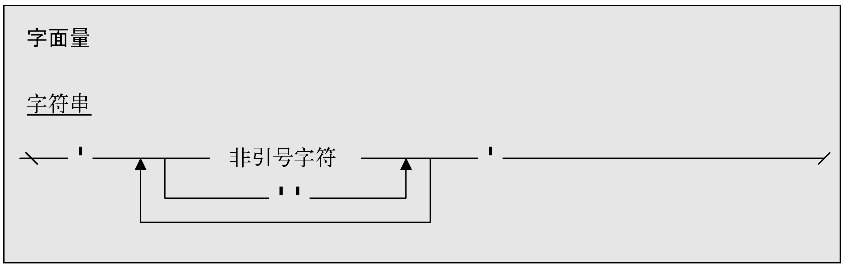


图 5-2　字符串字面量的语法图

下面是几个字符串字面量：

```
'This is a sample character string literal.'
'Here's yet another! '
'B-28'
'Seattle'
```

你可能注意到了，在图 5-1 和前面的第二个示例中，有一个看起来像双引号的东西。实际上，这不是双引号，而是两个相连的单引号。SQL 标准规定，要在字符串中嵌入单引号，可用两个相连的单引号表示。SQL 标准这样规定旨在让数据库系统能够将标识字符串字面量开头和结尾的单引号同包含在字面量中的单引号区分开来。下面的例子演示了其中的工作原理：

```
SQL            'The Vendor's name is:'
Displayed as   The Vendor's name is:
```

前面说过，可使用字符串字面量来改进 SELECT 语句返回的信息。虽然结果集中的信息通常很容易理解，但可能还能更清晰。例如，如果你执行下面的 SELECT 语句，结果集将只显示供应商的网站和名称：

```
SQL            SELECT VendWebPage, VendName
               FROM Vendors
```

在有些情况下，可定义一个字符串来提供额外的描述性文本，并将其添加到 SELECT 子句中，从而让信息更清晰。

请务必明智而审慎地使用这种方法，因为这种字符串字面量将出现在结果集的每一行中。下面演示了如何使用字符串字面量来修改前一个示例：

```
SQL            SELECT VendWebPage, 'is the Web site for',
                  VendName
               FROM Vendors
```

这条 SELECT 语句生成的结果集中的行类似于下面这样：

www.viescas.com	is the Web site for	John Viescas Consulting

这对网址做出了界定，让结果集中的信息更清晰些。这个示例虽然简单，但演示了可使用字符串字面量来做什么。本章后面将介绍如何在表达式中使用字符串字面量。

❖ **说明**　使用包含晦涩列名的遗留数据库时，这种方法很有用，但使用按第 2 章的建议创建的数据库时，不会经常用到这种方法。

5.4.2　数值字面量

在 SELECT 语句中可使用的另一种字面量是**数值字面量**。顾名思义，数值字面量由可选的符号和数字组成，可包含小数点、指数符号和指数。图 5-3 是数值字面量的语法图。

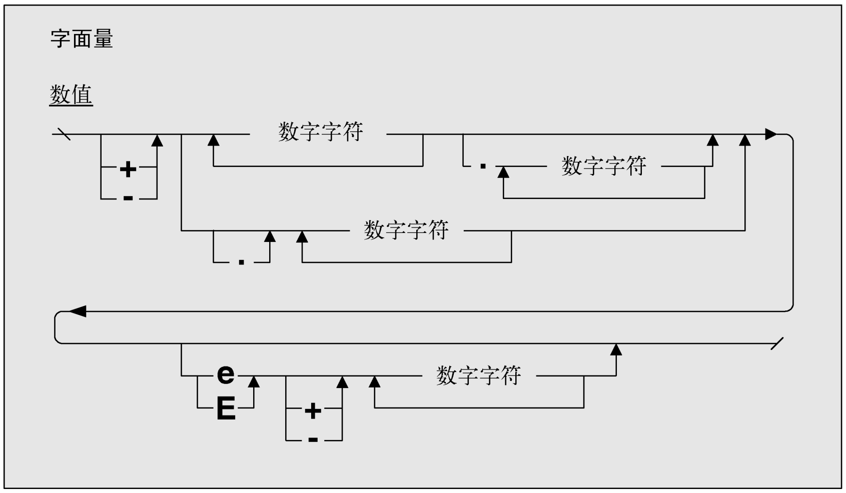


图 5-3　数值字面量的语法图

下面是一些数值字面量示例：

```
427
-11.253
.554
0.3E-3
```

数值字面量的最大用武之地是表达式（例如，乘以或加上一个固定的数值），因此将推迟到本章后面再讨论。

5.4.3　日期时间字面量

在 SELECT 语句中，可使用**日期字面量**、**时间字面量**和**时间戳字面量**来指定特定的日期和时间。SQL 标准将这些字面量统称为**日期时间字面量**。这些字面量定义起来很简单，如图 5-4 所示。

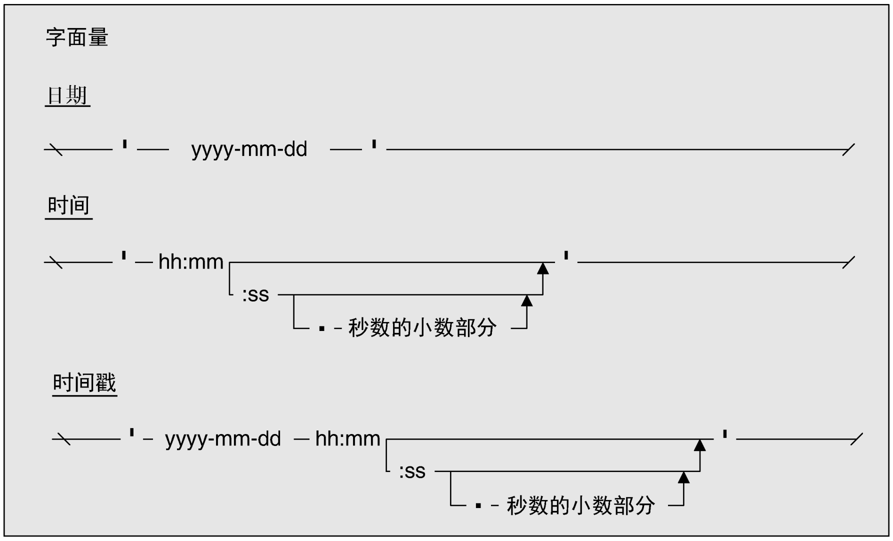


图 5-4　日期和时间字面量的语法图

使用日期时间字面量和间隔字面量时，请牢记以下几点。

- **日期**：日期字面量的格式为年-月-日，贯穿本书都将采用这种格式。然而，很多 SQL 数据库都支持更常用的格式月/日/年（美国）和日/月/年（除美国外的其他大多数国家）。SQL 标准还规定在日期字面量前必须加上关键字 DATE，但几乎所有的商用实现都允许只指定用分界符（通常是单引号）括起的字面量。我发现，在 MySQL 数据库系统中，要将日期字面量用于日期计算，必须将其用引号括起，并使用 CAST 函数将其转换为 DATE 数据类型。Microsoft Office Access 要求将井号（#）用作日期字面量分界符。
- **时间**：小时格式是基于 24 小时制的，例如，07:00 P.M.表示为 19:00。SQL 标准还规定在时间字面量前必须加上关键字 TIME，但几乎所有的商用实现允许只指定用分界符（通常是单引号）括起的字面量。我发现，在 MySQL 数据库系统中，要将时间字面量用于时间计算，必须将其用引号括起，并使用 CAST 函数将其转换为 TIME 数据类型。Microsoft Office Access 要求将井号（#）用作时间字面量分界符。
- **时间戳**：时间戳字面量就是由日期和时间组成的字面量，其中时间和日期之间用一个空格分隔。在时间戳中，日期和时间分别遵守有关日期和时间字面量的格式规则。SQL 标准还规定在时间戳字面量前必须加上关键字 TIMESTAMP，但几乎所有支持数据类型 TIMESTAMP 的商用实现允许只指定用分界符（通常是单引号）括起的字面量。

❖ **说明**　在有些数据库系统中，还可在表达式中使用日期时间字面量来定义间隔字面量，但本书不介绍这种字面量。有关这方面的详细信息，请参阅你使用的数据库系统的文档。

有关 SQL 标准定义的 DATE、TIME、TIMESTAMP 和 INTERVAL 的语法图，请参阅附录 A。

下面是几个日期时间字面量：

```
'2007-05-16'
'2016-11-22'
'21:00'
'03:30:25'
'2008-09-29 14:25:00'
```

请注意，在 MySQL 中，对于包含日期、时间或日期和时间的字符字面量，必须使用 CAST 函数显式地进行转换，如下面几个示例所示：

```
CAST('2016-11-22' AS DATE)
```

```
CAST('03:30:25' AS TIME)
CAST('2008-09-29 14:25:00' AS DATETIME)
```

前面说过，为遵循 SQL 标准，必须在字面量前面使用关键字指定其数据类型。SQL 标准规定，定义日期和时间字面量时，必须分别使用关键字 DATE 和 TIME，但大多数数据库系统不支持在这种环境下使用这些关键字，而只要求指定字面量的字符串部分。因此，在本书后面的所有示例中，对于日期或时间字面量，我将不再使用这些关键字，而使用单引号来界定它们。本章后面将介绍如何使用日期和时间表达式。有关如何按 SQL 标准定义日期时间字面量的更详细信息，请参阅附录 A。

5.5 表达式类型

编写 SQL 语句时，通常会用到下面 3 种表达式。

- **拼接表达式**：将多个字符列或字符字面量合并成一个字符串。
- **数学表达式**：对数值列或数值字面量执行加减乘除运算。
- **日期和时间算术运算表达式**：对日期和时间执行加减运算。

5.5.1 拼接表达式

SQL 标准规定拼接运算符为两条竖线。要拼接两个字符串，可将它们分别放在拼接运算符的两边，结果为两个字符串合并而成的字符串。图 5-5 是拼接表达式的语法图。

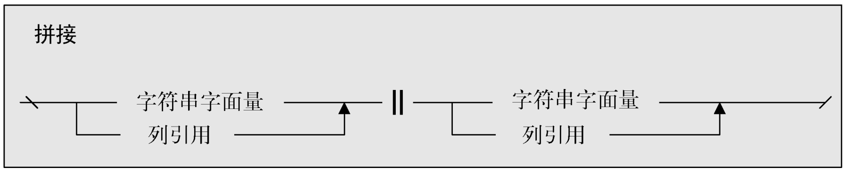


图 5-5 拼接表达式的语法图

❖ **说明** 在主要数据库系统中，我发现只有 IBM 的 DB2 和 Informix、Oracle 的 Oracle 以及 PostgreSQL 支持 SQL 标准定义的拼接运算符。Microsoft Office Access 支持拼接运算符&和+，Microsoft SQL Server 和 Ingres 支持+，而在 MySQL 中必须使用 CONCAT 函数。在本书所有的示例中，都将使用 SQL 标准定义的拼接运算符||。在本书配套网站中的示例数据库中，根据数据库类型（Microsoft Access、Microsoft SQL Server、MySQL 和 PostgreSQL）使用了相应的运算符。

拼接运算的基本工作原理如下。

表达式： `ColumnOne || ColumnTwo`

结果： ColumnOne 的内容 ColumnTwo 的内容

先来看世上最简单的示例——拼接两个字符串字面量，如名和姓。

表达式： `'Mike' || 'Hernandez'`

结果： MikeHernandez

对于这个示例，有两点需要注意。首先，姓和名都必须用单引号括起，因为它们是字符串字面量；其次，姓和名紧贴在一起。虽然拼接运算正确地合并了它们，但结果可能并非是你想要的。解决方案是再插入一个只包含一个空格的字符串字面量，从而在姓和名之间加上一个空格。

表达式： `'Mike' || ' '  || 'Hernandez'`

结果： Mike Hernandez

这个示例表明，要拼接更多的字符值，可使用更多的拼接运算符。对于可拼接的字符值数量没有任何限制，但对于拼接运算返回的字符串长度是有限制的。一般而言，拼接运算返回的字符串的长度不能超过变长字符数据类型支持的最大长度。你使用的数据库系统处理这种问题的方式可能稍有不同，因此更详细的信息请参阅文档。

拼接多个字符串合情合理，但也可以以同样的方式拼接多个字符列的值。例如，假设有两个列，分别名为 CompanyName 和 City，你可创建一个表达式，在其中使用列名将这两列的值拼接起来。下面是一个示例，它拼接了这两列的值和一个字符串。

表达式：`CompanyName || ' is based in ' || City`

结果：　DataTex Consulting Group is based in Seattle

无须将 CompanyName 和 City 用引号括起，因为它们是列引用（还记得前一章介绍的列引用吗？）。正如你将在本书后面的示例中看到的，在任何一种表达式中，都可使用列引用。

请注意，前面所有的拼接示例拼接的都是字符串。你可能会问：如果要拼接数字或日期，需要做特殊处理吗？在这方面，大多数数据库系统给你留了余地。看到你试图将字符列（或字面量）与数字或日期拼接时，数据库系统会自动对数字或日期进行类型转换，让拼接运算涉及的数据类型是兼容的。

但你不应依赖于数据库系统来悄悄地替你执行转换。要将字符串字面量或字符列的值与日期字面量或数值或日期列的值进行拼接，请使用 CAST 函数将数值或日期值转换为字符串。在下面的示例中，使用了 CAST 来转换日期列 DateEntered 的值。

表达式：`EntStageName || ' was signed with our agency on ' || CAST(DateEntered as CHARACTER(10))`

结果：　Modern Dance was signed with our agency on 1995-05-16

> ❖ **说明**　我显式地指定了 CHARACTER 数据类型的长度，这是因为 SQL 标准规定，指定长度时默认为 1。我发现，大多数主要实现在这方面给你留了余地，它们会生成足够长的字符串，以便包含转换得到的所有内容。有关这方面的详情，请参阅你使用的数据库系统的文档；但如果心存疑惑，可显式地指定长度。

另外，拼接数值字面量或数值列的值时，也应使用 CAST 函数将其转换为字符数据类型。在下面的示例中，使用了 CAST 来转换数值列 RetailPrice 的值：

表达式：`ProductName || ' sells for ' || CAST(RetailPrice AS CHARACTER(8))`

结果：　Trek 9000 Mountain Bike sells for 1200.00

在拼接表达式中，可同时包含字符串、日期时间值和数值。下面的示例演示了如何在表达式中同时使用这 3 种数据类型。

表达式：`' Order Number ' || CAST(OrderNumber AS CHARACTER(2)) || ' was placed on ' || CAST (OrderDate AS CHARACTER(10))`

结果：　Order Number 1 was placed on 2017-09-02

> ❖ **说明**　SQL 标准定义了很多函数，可用来从列中提取信息或根据一系列行计算出一个值，这将在第 12 章详细介绍。大多数商用数据库系统还提供了其他的函数，可用来操作字符串的组成部分以及设置日期、时间或金额的格式。有关这方面的详情，请参阅数据库系统文档。

介绍如何将来自各种来源的数据拼接成一个字符串后，下面来看看使用数值数据可创建的各种表达式。

5.5.2　数学表达式

SQL 标准定义了可对数值数据执行的加减乘除运算，还定义了用于计算绝对值、余数、乘方、对数等的常用函数。下表列出了 SQL 标准定义的数学函数。

函　　数	作　　用
ABS(<numeric expression>)	返回表达式的绝对值
MOD(<dividend>, <divisor>)	返回被除数除以除数得到的余数
LN(<numeric expression>)	返回表达式的自然对数
EXP(<numeric expression>)	返回自然对数的表达式次幂
POWER(<numeric base>, <numeric exponent>)	返回底数的指数次幂
SQRT(<numeric expression>)	返回表达式的平方根
FLOOR(<numeric expression>)	返回不大于表达式的最大整数
CEIL(<numeric expression>) CEILING(<numeric expression>)	返回不小于表达式的最小整数
WIDTH_BUCKET(<numeric value>, <numeric lower bound>, <numeric upper bound>, <numeric bucket count>)	将上限和下限指定的范围划分成 count 个区间，并返回 0 到 count + 1 之间的一个整数，以指出第一个参数位于哪个区间内

大多数 RDBMS 程序提供了这些函数，还提供了各种科学函数、三角函数、统计函数和数学函数。但本书只涉及 SQL 标准定义的 4 项基本运算——加减乘除。

编写数学表达式时，这 4 项基本数学运算的执行顺序（被称为**优先级**）很重要。SQL 标准规定，乘法和除法的优先级相同，且它们的优先级都高于加法和减法。这可能与你在学校学到的优先级（先乘后除，再加，然后执行减法）稍有不同，但与大多数现代编程语言采用的优先级一致。数学表达式是按从左到右的顺序计算的，这可能导致有趣的结果（具体取决于表达式是如何编写的）。有鉴于此，我强烈建议在复杂的数学表达式中大量使用括号，确保它们得以正确地计算。

如果你还记得在学校是如何编写表达式的，就知道如何在 SQL 中创建它们了。大体而言，你使用一个可带正负号的数值、一个数学运算符和另一个可带正负号的数值来创建表达式。图 5-6 演示了这个过程。

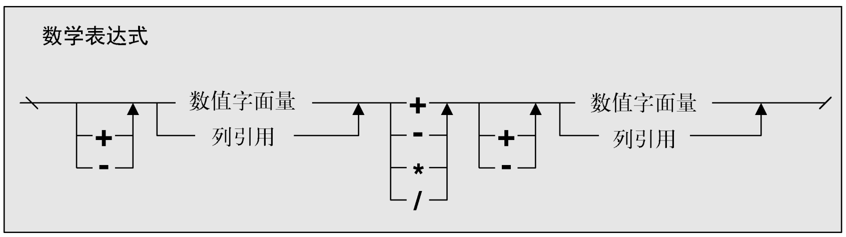


图 5-6　数学表达式的语法图

下面是一些数学表达式，它们分别使用了数值字面量、列引用以及它们的组合：

```
25 + 35
–12 * 22
RetailPrice * QuantityOnHand
TotalScore / GamesBowled
RetailPrice – 2.50
TotalScore / 12
```

前面说过，需要使用括号来确保复杂的数学表达式得以正确地计算。下面是一个简单的示例，演示了如何在这样的表达式中使用括号。

表达式： (11 – 4) +(12 * 3)

结果：　43

务必特别注意表达式中括号的位置，因为这将影响表达式的结果。下面示例中的两个表达式清楚地证明了这一点。虽然这两个表达式包含的数字和运算符相同，但括号的位置完全不同，这导致两个表达式的结果截然不同。

表达式： (23 * 11) + 12

结果：　265

表达式：`23 *(11 + 12)`

结果：　529

需要小心放置括号的原因显而易见，但不要因此而不使用括号。编写复杂的表达式时，括号很有用。

还可在表达式中使用括号来嵌套运算。当你使用括号来嵌套运算时，数据库系统将按从左到右、从内到外的顺序计算表达式。下面的表达式使用了括号来嵌套运算。

表达式：`(12 *(3 + 4)) - (24 / (10 +(6 - 4)))`

结果：　82

执行表达式中的运算并不像看起来那么难。下面列出了数据库系统计算上述表达式的顺序：

```
(3 + 4) = 7
(12 * 7) = 84      （将第一个运算的结果乘以 12）
(6 - 4) = 2
(10 + 2) = 12      （将第三个运算的结果加上 10）
(24 / 12) = 2      （将 24 除以第四个运算的结果）
84 - 2 = 82        （将 84 减去第二个运算的结果）
```

如你所见，数据库系统按从左到右的顺序计算，到遇到括号时先计算里面的表达式。也就是说，`(12 *(3 + 4))`和`(24 /(10 +(6 - 4)))`位于相同的层级，因此数据库系统先按从内到外的顺序计算左边的表达式，接下来它遇到了用括号括起的第二个表达式，因此按从内到外的顺序计算它。最后的运算是将左边的表达式的结果减去右边的表达式的结果（你头晕了吗？反正我是头晕了！）。

虽然前一个示例使用的是数值字面量，但完全可以使用列引用或数值字面量和列引用的组合。这里需要牢记的要点是，务必仔细地规划和定义数学表达式，确保它们返回期望的结果。请使用括号明确地指定你希望运算按什么样的顺序执行，这样就能得到期望的结果。

编写数学表达式时，务必确保其中的值是兼容的。在表达式包含列引用时，这一点尤其重要。为此，可像在拼接表达式中那样使用 CAST 函数。例如，假设有一个数据类型为 INTEGER 的 TotalLength 列，其中包含整数值 345，还有一个数据类型为 REAL 的 Distance 列，其中包含值 138.65。要将这两列的值相加，应使用 CAST 函数将 Distance 列的值转换为数据类型 INTEGER，或者将 TotalLength 列的值转换为数据类型 REAL（具体如何选择，取决于你希望最终结果的数据类型为 INTEGER 还是 REAL）。假设你希望结果为整数，可使用下面的表达式：

表达式：`TotalLength + CAST(Distance AS INTEGER)`

结果：　483

结果与你期望的不同？这可能是因为你认为将 138.65 转换为整数时，将向上取整。SQL 标准指出，使用 CAST 函数进行转换时如何取整由数据库系统决定，但大多数数据库系统在转换为整数时都截除小数部分。因此，这里假设我使用的数据库系统也是这样做的，即将 345 加上 138（而不是向上取整结果 139）。

如果你忘记了确保表达式中列值的兼容性，数据库系统可能引发错误。在这种情况下，数据库系统可能还会撤销运算。大多数 RDBMS 系统会自动处理转换，而不发出警告，但它们通常将所有的数字都转换为最复杂的数据类型，再计算表达式。在前面的示例中， RDBMS 很可能会将 TotalLength 转换为 REAL（两种数据类型中较复杂的那种），这是因为所有的 INTEGER 值都可包含在 REAL 数据类型中。然而，这可能不是你想要的结果。不自动执行转换的 RDBMS 程序通常会指出数据类型不匹配问题，让你知道需要如何修正表达式。

如你所知，只要做些规划并知道如何使用 CAST 函数，编写数学表达式就不会太难。下面将讨论如何编写加减日期和时间的表达式。

5.5.3　日期和时间算术表达式

SQL 标准定义了可对日期和时间执行的加法和减法运算。与你预期的相反，很多 RDBMS 程序都以不同的方式实

现这些运算。在有些数据库系统中，可像在数学表达式中那样定义这些运算，而在其他数据库系统中，必须使用特殊的内置函数来完成这种任务。有关你使用的数据库系统是如何处理这些运算的，请参阅其文档。本书只笼统地讨论日期和时间表达式，让你大致了解这些运算的工作原理。

1. 日期表达式

图 5-7 说明了 SQL 标准中定义的日期表达式的语法。如你所见，日期表达式创建起来非常简单，只需将两个值相加或相减即可。

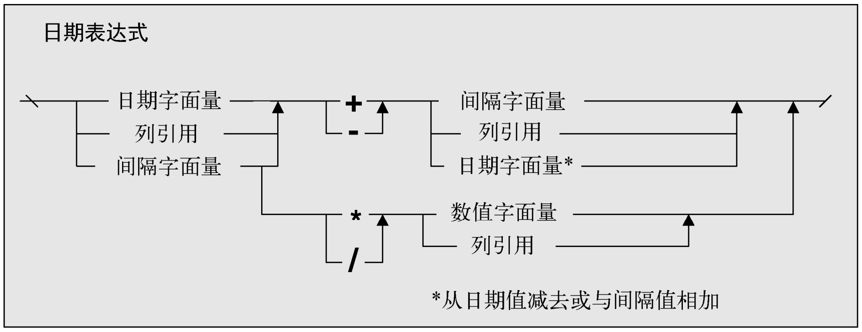


图 5-7　日期表达式的语法图

SQL 标准还定义了合法运算及其结果：

- 日期值减去或加上间隔值，结果为日期值；
- 日期值减去日期值，结果为间隔值；
- 间隔值加上日期值，结果为日期值；
- 间隔值加上或减去间隔值，结果为间隔值；
- 间隔值乘以或除以数字，结果为间隔值。

请注意，在 SQL 标准中，减数为日期值时，被减数只能是日期值，而加数为日期值时，被加数只能是间隔值。

使用列引用时，必须确保它为日期数据类型或间隔数据类型。如果加上或减去的列不是可接受的数据类型，可能必须使用 CAST 函数对其进行转换。SQL 标准明确规定，对于这些运算，只能使用指定的数据类型，但很多数据库系统会自动转换列的数据类型。是否必须显式地进行转换，取决于你使用的 RDBMS，因此请参阅其文档。

虽然只有为数不多的几种商用数据库系统支持间隔数据类型，但很多数据库系统都允许将日期值与整数值（如 SMALLINT 或 INT）相加减。你可将这种运算视为加上或减去指定的天数，这让你能够回答诸如“9 天后是什么日期”和“5 天前是什么日期”等问题。另外请注意，在有些数据库系统中，可将日期时间值加上或减去一个小数。例如，在 Microsoft Access 中，加上 3.5 的意思是加上 3 天零 12 小时。

将两个日期相减时，就是计算这两个日期的间隔。例如，你可能需要将当前日期减去聘用日期，以确定员工在公司工作了多久。虽然 SQL 标准规定，被加数为日期时，加数只能是间隔，但很多数据库系统（尤其是那些不支持间隔数据类型的数据库系统）都允许加数为数字或日期。你可使用这种计算来回答诸如“员工的下一次审核日期是什么时候”等问题。

❖ **说明**　SQL 标准定义了很多函数，可用来从列中提取信息或根据一系列行计算出一个值，这将在第 12 章详细介绍。大多数商用数据库系统还提供了其他的函数，可用来操作字符串的组成部分以及设置日期、时间或金额的格式。有关这方面的详情，请参阅数据库系统文档。

本书将演示简单的日期计算，并假定在你使用的数据库系统中，至少能够给日期值加上天数。还将假定将两个日期相减时，将得到一个整数，指出这两个日期相隔多少天。使用这些简单的概念，可创建你需要的大部分日期表达式。下面是你可以定义的一些日期表达式类型：

```
'2017-05-16' - 5
'2017-11-14' + 12
ReviewDate + 90
EstimateDate - DaysRequired
'2017-07-22' - '2017-06-13'
ShipDate - OrderDate
```

2. 时间表达式

你也可以使用时间值来创建表达式，其语法如图 5-8 所示。时间表达式很像日期表达式，适用于日期表达式的规则和限制也适用于时间表达式。

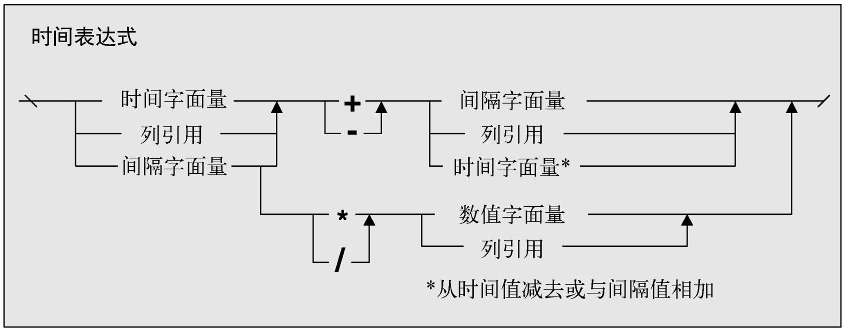


图 5-8　时间表达式的语法图

SQL 标准定义了合法运算及其结果：

- 时间值加上或减去间隔值，结果为时间值；
- 时间值减去时间值，结果为间隔值；
- 间隔值加上或减去间隔值，结果为间隔值；
- 间隔值乘以或除以数字，结果为间隔值。

请注意，在 SQL 标准中，减数为时间值时，被减数只能是时间值，而加数为时间值时，被加数只能是间隔值。

前述有关日期表达式的注意事项也适用于时间表达式。另外，在支持 DATETIME 数据类型的数据库系统中，这种值的时间部分被存储为以小数表示的天数，且至少精确到秒。在支持 DATETIME 数据类型的数据库系统中，通常还可给这种值加上或减去一个小数。例如，0.25 相当于 6 小时（四分之一天）。本书假设你使用的数据库系统支持加上和减去时间字面量或时间列，但不假定它们支持加上或减去小数。同样，有关你使用的数据库系统对时间表达式的实际支持情况，请参阅其文档。

基于上面的假设，下面是一些合法的时间表达式：

```
'14:00' + '00:22'
'19:00' - '16:30'
StartTime + '00:19'
StopTime - StartTime
```

前面说过，本书只笼统地介绍日期和时间表达式，目标是确保你从概念上理解日期和时间表达式，并大致知道你能够创建哪些类型的表达式。遗憾的是，大多数数据库系统没有完全遵循 SQL 标准中的时间表达式规范，且很多只部分支持日期表达式规范。然而，正如前面指出的，所有数据库系统都提供了可用来处理日期和时间的函数，附录 C 总结了 6 种主要实现中的这些函数。强烈推荐你研究你使用的数据库系统的文档，了解它都提供了哪些类型的函数。

知道如何编写各种表达式后，下一步是学习如何使用它们。

> ❖ **说明**　请参阅附录 C，其中概述了 6 种最流行的数据库系统是如何处理日期和时间的，还列出了它们支持的数据类型和算术运算以及它们提供的所有日期和时间函数。

5.6　在 SELECT 子句中使用表达式

知道如何使用表达式无疑是你将在本书中学习的最重要的技能之一。使用 SQL 时，你将使用表达式来完成各种任务。例如，你将使用表达式来完成如下任务：

- 在查询中创建计算列；
- 查找特定的列值；
- 筛选结果集中的行；
- 在 JOIN 操作中连接两个表。

在本书余下的篇幅中，我将演示如何使用表达式（并介绍其他技能）。首先来介绍如何在 SELECT 子句中使用表达式。

> ❖ **说明**　贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”方法。

通过在 SELECT 子句中使用简单表达式，可让结果集中的信息更清晰，还可让结果集包含更多的信息。例如，可使用表达式来拼接名和姓、计算商品总价、确定项目需要多长时间完成或指定患者下次就诊的时间。我们来看看如何在 SELECT 子句中使用拼接表达式、数学表达式和日期表达式。先来看看如何使用拼接表达式。

5.6.1　使用拼接表达式

不同于数学表达式和日期表达式，拼接表达式只能用于提高 SELECT 语句的结果集中信息的可读性。假设你要发出如下请求：

“给我一个清单，其中列出所有的员工及其电话号码。”

将这个请求转换为 SELECT 语句时，通过将名和姓合并为一列，可在一定程度上改善结果集。下面演示了一种对这个请求进行转换的方式。

转换	Select the first name, last name, and phone number of all our employees from the employees table（从员工表中选择所有员工的名、姓和电话号码）
整理	Select ~~the~~ first name, last name, ~~and~~ phone number ~~of all our employees~~ from ~~the~~ employees ~~table~~

SQL	`SELECT EmpFirstName || ' ' || EmpLastName,`
	`  'Phone Number: ' || EmpPhoneNumber`
	`FROM Employees`

结果集中的行将类似于下面这样：

Mary Thompson	Phone Number: 555-2516

你可能注意到了，除拼接名列、空格和姓列外，这里还拼接了字符串字面量“Phone Number:”和电话号码列。这个示例清楚地表明，可在 SELECT 子句中轻松地使用多个拼接表达式，以改善结果集中信息的可读性。前面说过，通过使用 CAST 函数，可拼接数据类型不同的值。例如，下面的示例拼接了字符串列和数值列：

“列出所有的供应商及其 ID。”

转换	Select the vendor name and vendor ID from the vendors table（从供应商表中选择供应商的名称和 ID）
整理	Select ~~the~~ vendor name ~~and~~ vendor ID from ~~the~~ vendors ~~table~~

SQL	`SELECT 'The ID Number for ' || VendName ||`
	`    ' is ' || CAST(VendorID AS CHARACTER)`
	`FROM Vendors`

编写 SELECT 语句时，拼接表达式是一个很有用的工具，但应明智而审慎地使用。别忘了，如果你使用的拼接表达式包含很长的字符串字面量，它将出现在结果集的每行中，这可能导致最终的结果充斥着重复的信息，根本起不到改善的作用。在拼接表达式中使用字面量时，一定要三思而后行，确保这样做是有益的。

5.6.2　给表达式命名

在 SELECT 子句中使用表达式时，结果集将包含新的列，其中显示的是表达式定义的运算的结果。这种新增的列称为计算（派生）列。例如，在下面的 SELECT 语句的结果集中，将包含 3 列——两个真正的列和一个计算列：

SQL

```
SELECT EmpFirstName || ' ' || EmpLastName,
   EmpPhoneNumber, EmpCity
FROM Employees
```

两个真正的列是 EmpPhoneNumber 和 EmpCity，而计算列是从 SELECT 子句开头的拼接表达式派生出来的。

SQL 标准指出，可使用关键字 AS 给计算列指定名称（实际上，可使用 AS 子句给任何列指定新的名称），但几乎所有数据库系统都要求必须给计算列指定名称。有些数据库系统要求你显式地提供名称，而有些会提供自动生成的名称。执行这里的示例前，请确定你使用的数据库系统是如何处理这一点的。如果打算在查询中引用表达式的结果，就应提供名称。

图 5-9 说明了给表达式命名的语法，你可使用任何合法的字符串字面量（用单引号括起）来指定名称。有些数据库系统放松了要求，仅当列名包含空格时才必须用引号括起，但我强烈建议你不要在名称中使用空格，因为在有些数据库编程语言中，空格处理起来很麻烦。

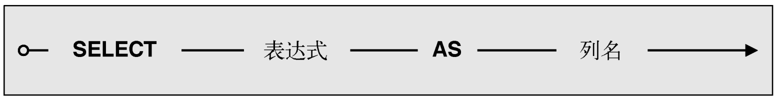


图 5-9　说明如何给表达式命名的语法图

下面来修改前一个示例中的 SELECT 语句，给拼接表达式指定名称：

SQL

```
SELECT EmpFirstName || ' ' || EmpLastName AS
    EmployeeName, EmpPhoneNumber, EmpCity
FROM Employees
```

这条 SELECT 语句的结果集将包含 3 列，分别名为 EmployeeName、EmpPhoneNumber 和 EmpCity。

除给表达式指定名称外，还可使用关键字 AS 来给真正的列提供别名。假设有一个名为 DOB 的列，而你担心有些用户可能不熟悉这个缩略语。为避免误读，可给这个列指定一个别名，如下所示。

SQL

```
SELECT EmpFirstName || ' ' || EmpLastName AS
    EmployeeName, DOB AS DateOfBirth
FROM Employees
```

这条 SELECT 语句生成的结果集包含两列，分别名为 EmployeeName 和 DateOfBirth。这有效地消除了结果集中的信息可能带来的迷惑。

给计算列提供名称对转换过程有细微的影响。例如，下面是前一个示例的一种可能的转换过程。

“列出所有员工的姓名和生日。”

转换	Select first name and last name as employee name and DOB as date of birth from the employees table（从员工表中选择姓、名和 DOB，并将姓和名显示为员工姓名，将 DOB 显示为生日）
整理	Select first name ~~and~~ || ‘ ’ || last name as EmployeeName ~~and~~ DOB as DateOfBirth from ~~the~~ employees ~~table~~

SQL

```
SELECT EmpFirstName || ' ' || EmpLastName
    AS EmployeeName, DOB AS DateOfBirth
FROM Employees
```

熟练使用表达式后，就无须像这里这样在转换结果中明确地指出它们，因为届时你将能够在编写 SELECT 语句时轻松地找出并定义它们。

❖ **说明** 在本书余下的篇幅中，只要情况合适，我就会给 SQL 语句中的计算列指定名称。

5.6.3 使用数学表达式

在 3 种表达式中，数学表达式可能是最“多才多艺”的，你可能经常用到。例如，可使用数学表达式来计算商品总价、确定一系列考试的平均成绩、计算两个实验结果的差、估算大楼的总容量。最为棘手的是确保表达式管用，但这只需仔细做点规划就成了。

下面的示例演示了如何在 SELECT 语句中使用数学表达式：

“显示每个经纪人的姓名和预期收入（薪水加提成，假定每位经纪人的销售额都是 50 000 美元）。”

转换	Select first name and last name as agent name and salary plus 50 000 times commission rate as projected income from the agents table（从经纪人表中选择名、姓和薪水，将名和姓显示为姓名，并将薪水加上 50 000 与佣金比例的乘积）
整理	Select first name ~~and~~ || ‘ ’ || last name as AgentName, ~~and~~ salary ~~plus~~ + 50000 ~~times~~ * commission rate as ProjectedIncome from ~~the~~ agents ~~table~~

SQL

```
SELECT AgtFirstName || ' ' || AgtLastName
     AS AgentName,
   Salary + (50000 * CommissionRate)
     AS ProjectedIncome
FROM Agents
```

请注意，为了清楚地指出我要先将 50 000 乘以佣金比例再与薪水相加（而不是先将薪水与 50 000 相加再乘以佣金比例），我使用了括号。从这个示例中可知，在同一条 SELECT 语句中，并非只能使用一种表达式。相反，可使用各种表达式来确保结果集中包含所需的信息。下面是另一种编写前述 SQL 语句的方式：

SQL

```
SELECT AgtFirstName || ' ' || AgtLastName
    || ' has a projected income of ' ||
    CAST(Salary + (50000 * CommissionRate)
      AS CHARACTER) AS ProjectedIncome
FROM Agents
```

使用数学表达式可提供的信息几乎是无限的，但必须妥善地规划表达式，并合理地使用 CAST 函数。

5.6.4 使用日期表达式

日期表达式的用法与数学表达式类似，也只需加上或减去值。可使用日期表达式来完成各种任务，例如，可计算预期的发货日期，估算项目需要多少天完成，确定患者的下次就医日期。下面的示例演示了如何在 SELECT 子句中使用日期表达式：

“各个订单在下单后多少天发货？”

转换	Select the order number and ship date minus order date as days to ship from the orders table（从订单表中选择订单号和发货日期，将发货日期减去订购日期，并将其显示为多少天后发货）
整理	Select ~~the~~ order number ~~and~~ ship date ~~minus~~ – order date as DaysToShip from ~~the~~ orders ~~table~~

SQL	`SELECT OrderNumber, CAST(ShipDate – OrderDate AS INTEGER) AS DaysToShip FROM Orders`

可像使用日期表达式那样使用时间表达式。

“如果所有课程都推迟 10 分钟，各个课程将在什么时间开始？”

转换	Select the start time and start time plus 10 minutes as new start time from the classes table（从课程表中选择开始时间，将开始时间加上 10 分钟，并显示为新的开始时间）
整理	Select ~~the~~ start time ~~and~~ start time ~~plus~~ + ‘00:10’ ~~minutes~~ as NewStartTime from ~~the~~ classes ~~table~~
SQL	`SELECT StartTime, StartTime + '00:10' AS NewStartTime FROM Classes`

前面说过，所有数据库系统都提供了用于处理日期值的函数。前面的介绍旨在让你大致了解如何在 SELECT 语句中使用日期和时间，这里再次推荐你参考你使用的数据库系统的文档，详细了解它提供的日期和时间函数。

5.6.5　说点题外话：值表达式

至此，你知道了如何在 SELECT 子句中使用列引用、字面量值和表达式，还知道了如何给列引用或表达式指定名称，下面来说说它们在更大的场景中所处的位置。

SQL 标准将列引用、字面量值和表达式统称为**值表达式**。图 5-10 说明了如何定义值表达式。

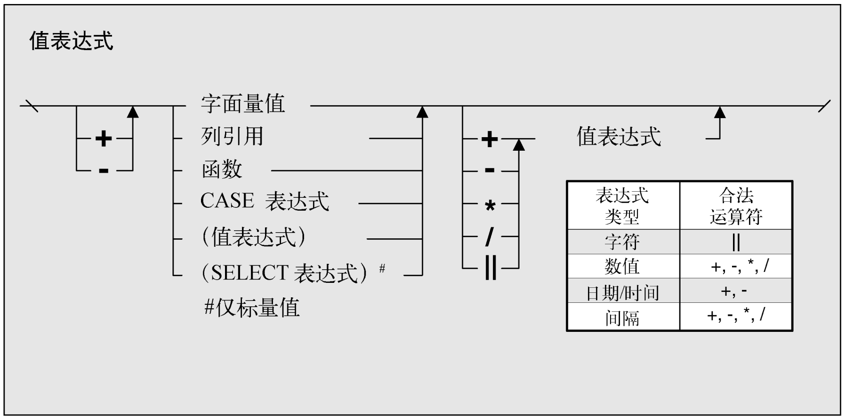


图 5-10　值表达式的语法图

下面来详细说说值表达式的组成部分。

- ❑ 值表达式以可选的加号或减号打头，你使用这些符号来让值表达式返回一个有符号数值。值本身可以是数值字面量、数值列的值、返回数值的函数调用（参见前面对 CAST 函数的讨论）或数学表达式的返回值。对于返回字符或日期时间数据类型的表达式，不能在它前面加上加号或减号。
- ❑ 如你所见，图 5-10 左边的列表中还包含（值表达式），这意味着可以使用复杂的值表达式，这种值表达式由本身包含拼接或数学运算符的值表达式组成。这里的括号让数据库系统先计算其中的值表达式［现在不用考虑（SELECT 表达式）和 CASE 表达式，这两种表达式将分别在第 11 章和第 19 章详细介绍］。
- ❑ 语法图中的下一项是一个运算符列表。从语法图中内嵌的表格可知，第一个表达式的类型决定了你可使用哪些运算符。

- 运算符列表后面还是一个值表达式，这意味着可在值表达式中包含其他值表达式，这让你能够创建非常复杂的表达式。

根据定义，值表达式返回一个值，供 SQL 语句的某个部分使用。SQL 标准规定，可在各种语句中使用值表达式，但不管在哪里使用值表达式，你都将以刚才介绍的方式定义它们。

为加深认识，下面来演示如何在 SELECT 语句中使用值表达式。图 5-11 是图 4-9 所示 SELECT 语句语法图的修订版，这个新版本提供了更大的灵活性，支持在同一条 SELECT 语句中使用字面量、列引用、表达式或它们的组合。还可使用关键字 AS 给值表达式命名。

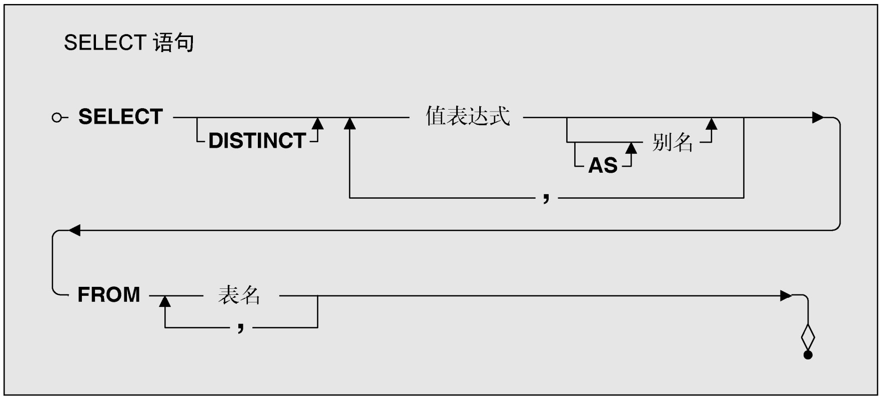


图 5-11　包含值表达式的 SELECT 语句的语法图

在本书余下的篇幅中，我将在合适的情况下使用术语**值表达式**来表示列引用、字面量值或表达式。本书后面将讨论如何在其他语句中使用值表达式，还有值表达式可表示的其他东西。

现在言归正传。

5.7　空值：Null

你知道，表由行和列组成，其中每列表示表表示的客体的一个特征，而每行表示表表示的客体的一个实例。你还可将行视为一组完整的列值——每行都包含表中各列的一个值。图 5-12 展示了一个典型的表。

Customers

CustomerID	CustFirstName	CustLastName	CustStreetAddress	CustCity	CustCounty	CustState
1001	Suzanne	Viescas	15127 NE 24th, #383	Redmond	King	WA
1002	William	Thompson	122 Spring River Drive	Duvall	King	WA
1003	Gary	Hallmark	Route 2, Box 203B	Auburn	King	WA
1004	Robert	Brown	672 Lamont Ave	Houston		TX
1005	Dean	McCrae	4110 Old Redmond Rd.	Redmond		WA
1006	John	Viescas	15127 NE 24th, #383	Redmond	King	WA
1007	Mariya	Sergienko	901 Pine Avenue	Portland		OR
1008	Neil	Patterson	233 West Valley Hwy	San Diego	San Diego	CA

图 5-12　典型的 Customers 表

到目前为止，本书介绍了如何使用 SELECT 语句从表中数据检索信息，以及如何使用值表达式来操作数据。这些做法之所以可行，是因为我一直假设表中的每列都包含数据，但图 5-12 清楚地表明，在表中的特定行中，有些列可能没有值。缺失值可能会给 SELECT 语句和值表达式带来负面影响，这取决于你是如何使用数据的。下面先来说说 SQL 是如何看待缺失值的，再来说说负面影响。

5.7.1 Null 简介

在 SQL 中，Null 表示值缺失或未知。首先，你必须明白，Null 表示的并非零、由空格组成的字符串或长度为零的字符串，其中的原因非常简单。

- 零的含义非常丰富，可表示账户余额、可升级到头等舱的机票数量或特定商品库存量。
- 虽然对大多数人来说，由空格组成的字符串毫无意义，但对 SQL 来说是绝对有意义的。在 SQL 看来，空格是合法的字符，因此由三个空格组成的字符串（'　'）与由多个字母组成的字符串一样（'a character string'），也是合法的。
- 在特定的情形下，长度为零的字符串（两个相连的单引号，中间没有空格，即''）是有意义的。例如，在员工表的 MiddleInitial 列中，长度为零的字符串可能表示相应的员工没有中间名。但请注意，有些实现（其中最著名的是 Oracle）将 VARCHAR 列中长度为零的字符串视为 Null。

在使用正确的情况下，Null 很有用，图 5-12 所示的 Customers 表很好地说明了这一点。在 CustCounty 列中，每个空单元格都表示县名缺失或未知，即为 Null。要正确地使用 Null，必须明白它最初是怎么出现的。

缺失值通常是人为错误导致的。例如，请看表示 Robert Brown 的行。如果你输入 Robert Brown 的数据时，忘记询问他居住在哪个县，这项数据将是缺失的，在相应的行中表示为 Null。然而，意识到这种错误后，你可致电 Robert Brown 并询问他居住在哪个县，以改正错误。

未知值出现在表中的原因很多，其中一个可能是要在列中包含的值还未定义。例如，School Scheduling 数据库中可能有一个 Categories 表，但对于即将在秋季开设的一组新课程，其所属的类别还未包含在这个表中。表中可能包含未知值的另一个原因是，这些值确实是未知的。请看图 5-12 所示 Customers 表中表示 Dean McCrae 的行。假设你在输入 Dean McCrae 的数据，你询问他居住在哪个县。如果你和他都不知道，那么相应行中 CustCounty 列的值就确实是未知的。这将在相应行中表示为 Null。显然，确定正确的县名后，就可修复这个问题。

如果在特定行中，某列不能有值，那么该列的值也可能为 Null。假设有一个员工表，其中包含 Salary 列和 HourlyRate 列。在这两列中，总有一列的值为 Null，因为对于同一名员工，其薪水不能既为固定的又按小时计薪。

必须指出的是，“没有值”和“不适用”之间存在细微的差别。在前一个示例中，两列中有一列的值是没有的（does not apply）。假设有一个病患表，其中包含一个 HairColor 列，而你要更新一名男性患者的行。如果这位患者是秃头，那么该列的值就是不适用的（is not applicable）。虽然可以使用 Null 来表示不适用的值，但建议使用真正的值，如 N/A 或 Not Applicable，因为从长远看，这将让信息更清晰。

如你所见，是否允许表中包含 Null 取决于你将如何使用数据。介绍使用 Null 的正面作用后，下面来看看它的负面影响。

5.7.2 Null 带来的问题

Null 的主要缺点在于它给数学运算带来的负面影响。任何包含 Null 的运算的结果都为 Null，这合乎逻辑（如果一个值未知，那么对其执行运算的结果必然也是未知的）。下面的示例演示了 Null 将如何影响运算的结果：

```
(25 * 3) + 4 = 79
(Null * 3) + 4 = Null
(25 * Null) + 4 = Null
(25 * 3) + Null = Null
```

运算包含值为 Null 的列时，结果也将如此。例如，假设你执行下面的 SELECT 语句（这里只是举例，这条语句在示例数据库中不管用），它将返回如图 5-13 所示的结果集。

```
SQL         SELECT ProductID, ProductDescription, Category,
                Price, QuantityOnHand, Price *
               QuantityOnHand AS TotalValue
            FROM Products
```

只要 Price 和 QuantityOnHand 列包含有效的数值，TotalValue 列表示的运算就能成功地执行；但只要 Price 和 QuantityOnHand 列有一个包含 Null，TotalValue 就将包含 Null。好消息是，将 Price 和 QuantityOnHand 列中的 Null 替换为有效的数值后，TotalValue 列将包含合适的值。通过确保在数学表达式中使用的列都不包含 Null 值，可从根本上避免这种问题。

ProductID	ProductDescription	Category	Price	QuantityOnHand	TotalValue
70001	Shur-Lok U-Lock	Accessories		12	
70002	SpeedRite Cyclecomputer		65.00	20	1,300.00
70003	SteelHead Microshell Helmet	Accessories	36.00	33	1,118.00
70004	SureStop 133-MB Brakes	Components	23.50	16	376.00
70005	Diablo ATM Mountain Bike	Bikes	1,200.00		
70006	UltraVision Helmet Mount Mirrors		7.45	10	74.50

图 5-13 包含在数学表达式中的 Null 值

对 Null 的讨论不会就此止步，第 12 章还将介绍 Null 对汇总信息的 SELECT 语句的影响。

5.8 语句举例

知道如何在 SELECT 语句的 SELECT 子句中使用各种值表达式后，我们来看一些示例，它们使用了 4 个示例数据库中的表。这些示例演示了如何使用表达式来生成输出列。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库中，查询名以 CH05 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

❖ **说明** 下面的示例将转换和整理合二为一了，这旨在让你能够整合这个过程。虽然在每章的正文中还是按三步走，但在“语句举例”一节，你将有机会练习合并后的流程。

- **Sales Orders 数据库**

“显示所有商品的库存价值。”

转换/整理	Select ~~the~~ product name, retail price ~~times~~ * quantity on hand as InventoryValue from ~~the~~ products ~~table~~（从商品表中选择商品名、零售价和库存量，并将零售价和库存量的乘积显示为库存价值）

SQL

```
SELECT ProductName,
    RetailPrice * QuantityOnHand AS
    InventoryValue
FROM Products
```

CH05_Product_Inventory_Value（40 行）

ProductName	InventoryValue
Trek 9000 Mountain Bike	$7,200.00
Eagle FS-3 Mountain Bike	$14,400.00
Dog Ear Cyclecomputer	$1,500.00
Victoria Pro All Weather Tires	$1,099.00
Dog Ear Helmet Mount Mirrors	$89.40
Viscount Mountain Bike	$3,175.00
Viscount C-500 Wireless Bike Computer	$1,470.00
Kryptonite Advanced 2000 U-Lock	$1,000.00
<< 其他行 >>	

“各个订单从下单到发货相隔多少天？”

转换/整理 Select ~~the~~ order number, order date, ship date, ship date ~~minus~~ - order date as DaysElapsed from ~~the~~ orders ~~table~~（从订单表中选择订单号、下单日期和发货日期，并将从下单到发货的天数显示为 DaysElapsed）

SQL
```
SELECT OrderNumber, OrderDate, ShipDate,
   CAST(ShipDate - OrderDate AS INTEGER)
   AS DaysElapsed
FROM Orders
```

CH05_Shipping_Days_Analysis（944 行）

OrderNumber	OrderDate	ShipDate	DaysElapsed
1	2017-09-02	2017-09-05	3
2	2017-09-02	2017-09-04	2
3	2017-09-02	2017-09-05	3
4	2017-09-02	2017-09-04	2
5	2017-09-02	2017-09-02	0
6	2017-09-02	2017-09-06	4
7	2017-09-02	2017-09-05	3
8	2017-09-02	2017-09-02	0
9	2017-09-02	2017-09-05	3
10	2017-09-02	2017-09-05	3
<< 其他行 >>			

- **Entertainment Agency 数据库**

“各个演出持续多长时间？”

转换/整理 Select ~~the~~ engagement number, end date ~~minus~~ - start date ~~plus one~~ + 1 as DueToRun from ~~the~~ engagements ~~table~~（从演出合约表中选择演出合约编号、结束日期和起始日期，并将起始日期到结束日期的天数显示为 DueToRun）

SQL
```
SELECT EngagementNumber,
    CAST(CAST(EndDate - StartDate
      AS INTEGER) + 1 AS CHARACTER)
   || ' day(s)' AS DueToRun
FROM Engagements
```

CH05_Engagement_Lengths（111 行）

EngagementNumber	DueToRun
2	5 day(s)
3	6 day(s)
4	7 day(s)
5	4 day(s)
6	5 day(s)
7	8 day(s)
8	8 day(s)
9	11 day(s)
10	10 day(s)
11	2 day(s)
<< 其他行 >>	

❖ **说明** 要计算演出的天数，必须在日期表达式中加 1，否则如果起始日期和结束日期是同一天，结果将为 0 天。另外，注意，为加上 1，我使用 CAST 将两个日期相减的结果转换成了 INTEGER（在 MySQL 中，为 Signed Integer），再将结果转换为 CHARACTER，以确保拼接操作的结果符合预期。

“各个演出合约的净收入是多少？”

转换/整理　Select ~~the~~ engagement number, contract price, contract price ~~times~~ * 0.12 as OurFee, contract price ~~minus~~ -（contract price ~~times~~ * 0.12）as NetAmount from ~~the~~ engagements ~~table~~（从演出合约表中选择演出合约号和合约价格，并将合约价格乘以 0.12 的结果显示为 OurFee，将合约价格减去合约价格乘以 0.12 的结果显示为 NetAmount）

SQL

```
SELECT EngagementNumber, ContractPrice,
   ContractPrice * 0.12 AS OurFee, ContractPrice
     -(ContractPrice * 0.12)
   AS NetAmount
FROMp Engagements
```

CH05_Net_Amount_Per_Contract（111 行）

EngagementNumber	ContractPrice	OurFee	NetAmount
2	$200.00	$24.00	$176.00
3	$590.00	$70.80	$519.20
4	$470.00	$56.40	$413.60
5	$1,130.00	$135.60	$994.40
6	$2,300.00	$276.00	$2,024.00
7	$770.00	$92.40	$677.60
8	$1,850.00	$222.00	$1,628.00
9	$1,370.00	$164.40	$1,205.60
10	$3,650.00	$438.00	$3,212.00
11	$950.00	$114.00	$836.00
<< 其他行 >>			

- **School Scheduling 数据库**

“计算各位教员到 2017 年 10 月 1 日在学校工作了多少个整年，并按姓和名对结果排序。”

转换/整理　Select last name || ‘,’ || ~~and~~ first name ~~concatenated with a comma~~ as Staff, date hired, ~~and~~((‘2017-10-01’ ~~minus~~ - date hired) ~~divided by~~ / 365) as YearsWithSchool from ~~the~~ staff ~~table and sort~~ order by last name ~~and~~ first name（从教员表中选择姓、名和聘用日期，将姓、逗号和名拼接起来并显示为 Staff，将聘用日期到 2017 年 10 月 1 日的天数除以 365，并将结果显示为 YearsWithSchool，再按姓和名排序）

SQL

```
SELECT StfLastName || ', ' || StfFirstName
      AS Staff,
    DateHired,
    CAST(CAST('2017-10-01' - DateHired
        AS INTEGER) / 365 AS INTEGER)
      AS YearsWithSchool
FROM Staff
ORDER BY StfLastName, StfFirstName
```

CH05_Length_Of_Service（27 行）

Staff	DateHired	YearsWithSchool
Alborous, Sam	1990-11-20	26
Black, Alastair	1996-12-11	20
Bonnicksen, Joyce	1994-03-02	23
Brehm, Peter	1994-07-16	23
Brown, Robert	1997-02-09	20
Coie, Caroline	1991-01-28	26
DeGrasse, Kirk	1996-03-02	21
Ehrlich, Katherine	1993-03-08	24
Glynn, Jim	1993-08-02	23
Hallmark, Alaina	1992-01-07	24
<< 其他行 >>		

❖ **说明**　这里的目标是计算到 2017 年 10 月 1 日工作了多少个整年（请注意，我使用了 CAST 函数来确保字符串字面量被视为日期）。例如，如果一名教员是 2015 年 10 月 10 日聘用的，答案应该是 1，而不是 2。从技术上说，这条 SELECT 语句中的表达式是正确的且像期望的那样工作，但如果在聘用日期和 2017 年 10 月 1 日之间有闰年，返回的结果将不正确。奇怪的是，SQL 标准没有定义任何执行日期和时间计算的函数，而只定义了两个日期/时间之间的减法运算、日期/时间和间隔的加法运算以及间隔与数字的乘除运算。

要修复上述问题，可使用数据库系统提供的合适的日期算术函数。前面说过，大多数数据库系统提供了处理日期和时间的方法，附录 C 简要地介绍了 6 种主要的数据库系统支持的日期和时间函数。使用这些函数时务必小心，例如，Microsoft SQL Server 和 Microsoft Office Access 都提供了函数 DateDiff，让你能够计算两个日期相差的年数，但它返回的结果为两个日期的年份部分的差。因此，在这个函数看来，2016 年 12 月 31 日和 2017 年 1 月 1 日之间相隔 1 年！第 19 章将演示如何使用 CASE 更准确地回答这种问题。

“给我提供一个清单，列出所有教员的姓名、薪水和奖金（薪水的 7%）。”

转换/整理　Select ~~the~~ last name || ‘,’ || ~~and~~ first name as StaffMember, salary, ~~and~~ salary ~~times~~ * 0.07 as Bonus from ~~the~~ staff ~~table~~（从教员表中选择姓、名和薪水，将姓、逗号和名拼接起来并显示为 StaffMember，将薪水乘以 0.07 的结果显示为 Bonus）

SQL
```
SELECT StfLastName || ', ' || StfFirstName
    AS Staff, Salary, Salary * 0.07 AS Bonus
FROM Staff
```

CH05_Proposed_Bonuses（27 行）

Staff	Salary	Bonus
Alborous, Sam	$60,000.00	$4,200.00
Black, Alastair	$60,000.00	$4,200.00
Bonnicksen, Joyce	$60,000.00	$4,200.00
Brehm, Peter	$60,000.00	$4,200.00
Brown, Robert	$49,000.00	$3,430.00
Coie, Caroline	$52,000.00	$3,640.00
DeGrasse, Kirk	$45,000.00	$3,150.00
Ehrlich, Katherine	$45,000.00	$3,150.00
Glynn, Jim	$45,000.00	$3,150.00
Hallmark, Alaina	$57,000.00	$3,900.00
<< 其他行 >>		

- **Bowling League 数据库**

“显示所有的投球手和地址（设置为适合用作邮寄清单的格式），并按邮政编码排序。”

转换/整理　Select first name || ‘ ’ || ~~and~~ last name as FullName, BowlerAddress, city || ‘,’ || state || ‘ ’ || ~~and~~ ZIP Code as CityStateZip, BowlerZip from ~~the~~ bowlers ~~table and~~ order by ZIP Code（从投球手表中选择姓、名、地址、城市、州和邮政编码，将姓、逗号和名拼接起来并显示为 FullName，将城市、逗号、州、空格和邮政编码拼接起来并显示为 CityStateZip，再按邮政编码排序）

SQL
```
SELECT BowlerFirstName || ' ' || BowlerLastName AS
    FullName,
   Bowlers.BowlerAddress,
   BowlerCity || ', ' || BowlerState || ' ' ||
   BowlerZip AS CityStateZip, BowlerZip
FROM Bowlers
ORDER BY BowlerZip
```

CH05_Names_Address_For_Mailing（32 行）

FullName	BowlerAddress	CityStateZip	BowlerZip
Kathryn Patterson	16 Maple Lane	Auburn, WA 98002	98002
Rachel Patterson	16 Maple Lane	Auburn, WA 98002	98002
Ann Patterson	16 Maple Lane	Auburn, WA 98002	98002
Neil Patterson	16 Maple Lane	Auburn, WA 98002	98002
Megan Patterson	16 Maple Lane	Auburn, WA 98002	98002
Carol Viescas	16345 NE 32nd Street	Bellevue, WA 98004	98004
Sara Sheskey	17950 N 59th	Seattle, WA 98011	98011
Richard Sheskey	17950 N 59th	Seattle, WA 98011	98011
William Thompson	122 Spring Valley Drive	Duvall, WA 98019	98019
Mary Thompson	122 Spring Valley Drive	Duvall, WA 98019	98019
<< 其他行 >>			

❖ **说明**　注意，我不仅将 BowlerZip 列包含在 CityStateZip 表达式中，还将其作为独立的列。本书前面说过，SQL 标准规定，只能按 SELECT 子句中包含的列进行排序。虽然无须再次加上 BowlerZip 列就能创建邮寄清单，但必须包含这一列，这样才能在 ORDER BY 子句中使用它。有些数据库（最著名的是 Microsoft Office Access）没有这样的要求，但别忘了在每个示例查询中，我都严格地遵循了 SQL 标准。

“显示各个投球手在各局比赛中让分得分和原始得分的差。”

转换/整理	Select bowler ID, match ID, game number, handicap score, raw score, handicap score ~~minus~~ - raw score as PointDifference from ~~the~~ bowler scores ~~table and~~ order by bowler ID, match ID, game number（从投球手得分表中选择投球手 ID、场次 ID、局次、让分得分和原始得分，将让分得分和原始得分的差显示为 PointDifference，并按投球手 ID、场次 ID 和局次排序）

SQL

```
SELECT BowlerID, MatchID, GameNumber,
   HandiCapScore, RawScore, HandiCapScore
    - RawScore AS PointDifference
FROM Bowler_Scores
ORDER BY BowlerID, MatchID, GameNumber
```

CH05_Handicap_vs_RawScore（1344 行）

BowlerID	MatchID	GameNumber	HandiCapScore	RawScore	PointDifference
1	1	1	192	146	46
1	1	2	192	146	46
1	1	3	199	153	46
1	5	1	192	145	47
1	5	2	184	137	47
1	5	3	199	152	47
1	10	1	189	140	49
1	10	2	186	137	49
1	10	3	210	161	49
<< 其他行 >>					

5.9　小结

本章首先简要地介绍了表达式，接着阐述了要编写表达式必须理解数据类型的原因，并详细讨论了主要的数据类型。接下来介绍了 CAST 函数，并指出经常需要使用它来修改列或字面量的数据类型，使其与要编写的表达式兼容。然后介绍了在表达式中引入常量（字面量）的各种方式，阐述了如何使用表达式来增减从数据库检索的信息量，还指

出了表达式就是涉及数字、字符串或日期和时间的运算。

接下来，讨论了各种类型的表达式，演示了如何拼接字符串以及如何使用 CAST 函数来拼接字符串和其他数据类型，还演示了如何编写数学表达式，并阐述了优先顺序对数学表达式的影响。这部分最后介绍了日期和时间表达式：先介绍 SQL 标准是如何处理日期和时间的，之后指出大多数数据库系统提供了处理日期和时间的方法。

然后，讨论了如何在 SELECT 语句中使用表达式，演示了如何在 SELECT 子句中插入表达式，如何在表达式中同时使用字面量值和列，以及如何给用于放置表达式结果值的列命名。这部分的最后简要介绍了值表达式，指出了 SQL 标准使用这个术语来统称列引用、字面量值和表达式，而你可在 SQL 语句的各种子句中使用值表达式（这将在本书后面详细介绍）。

最后，本章讨论了 Null。Null 表示值缺失或未知。这部分阐述了如何妥善地使用 Null，并指出了 Null 在合适的情况下很有用。另外，还讨论了 Null 给数学运算带来的负面影响。现在你知道，包含 Null 值的数学运算将返回 Null。此外还阐述了 Null 可能导致结果集中的信息不准确。

下一章将讨论如何检索一组特定的信息，演示如何使用 WHERE 子句来筛选 SELECT 语句检索到的信息。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

5.10　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**
  (1) 如果调整每种商品的价格，将其降价 5%，结果将如何？
  解决方案见 CH05_Adjusted_Wholesale_Prices（90 行）。
  (2) 给我一个清单，列出每位顾客下的订单，并按下单日期降序排列。
  提示：为正确地显示信息，可能需要根据多列进行排序。
  解决方案见 CH05_Orders_By_Customer_And_Date（944 行）。
  (3) 编写一个供应商清单，列出所有供应商的名称和地址，并按供应商名排序。
  解决方案见 CH05_Vendor_Addresses（10 行）。
- **Entertainment Agency 数据库**
  (1) 按城市列出所有顾客的姓名。
  提示：必须使用 ORDER BY 子句，并在其中指定一列。
  解决方案见 CH05_Customers_By_City（15 行）。
  (2) 列出所有的演唱组合及其网站。
  解决方案见 CH05_Entertainer_Web_Sites（13 行）。
  (3) 显示每位经纪人最初 6 个月的业绩考核日期。
  提示：要完成这个请求，需要使用日期算术。请务必参阅附录 C。
  解决方案见 CH05_First_Performance_Review（9 行）。
- **School Scheduling 数据库**
  (1) 给我一个按薪水降序排列的教员清单。
  解决方案见 CH05_Staff_List_By_Salary（27 行）。
  (2) 给我一个教员电话号码清单。
  解决方案见 CH05_Staff_Member_Phone_List（27 行）。
  (3) 列出所有学生的姓名，并按居住的城市排列。
  解决方案见 CH05_Students_By_City（18 行）。

- **Bowling League 数据库**

  (1) 显示下一年在每个联赛地点举行联赛的日期。

  提示：要确定下一年同一星期的同一天，可加上 364 天，请务必参阅附录 C。

  解决方案见 CH05_Next_Years_Tourney_Dates（20 行）。

  (2) 列出每个投球手的姓名和电话号码。

  解决方案见 CH05_Phone_List（32 行）。

  (3) 给我一个清单，其中列出每个球队的阵容。

  提示：使用 Bowlers 表来完成这个查询。

  解决方案见 CH05_Team_Lineups（32 行）。

第 6 章 筛选数据

“我有 6 名忠实的仆人（我所知道的一切都是他们教的），他们是‘何事’‘为何’‘何时’‘如何’‘何地’与‘何人’。”

——拉迪亚德 · 吉卜林，*I Keep Six Honest-serving Man*

本章涵盖如下主题：

- ❑ 使用 WHERE 提炼信息
- ❑ 定义查找条件
- ❑ 使用多个条件
- ❑ 再谈 Null：一个注意事项
- ❑ 以不同的方式表示条件
- ❑ 语句举例
- ❑ 小结
- ❑ 练习

前两章讨论了查看特定表中所有信息的方法，还讨论了如何创建并使用表达式来增减信息。本章将介绍如何使用 WHERE 子句来筛选信息，以调整检索到的内容。

6.1 使用 WHERE 提炼信息

到目前为止，使用的 SELECT 语句都检索特定表中的所有行并在结果集中显示它们。如果确实要查看表中的所有信息，这很好，但如果只想查找与特定的人物、地点、数值或日期范围相关的行，该怎么办呢？这样的要求很常见，实际上，向数据库提出的很多问题都有这样的要求。例如，你可能提出下面这样的问题。

“哪些顾客来自西雅图？”
“给我一个清单，列出贝尔维尤的所有员工及其电话号码。”
“当前我们提供哪些类型的音乐课程？”
“给我一个清单，列出学分为 3 的课程。”
“哪些演唱组合有网站？”
“给我一个清单，列出演唱组合 Caroline Coie Trio 的演出合约。”
“给我一个清单，列出在 5 月份下过单的顾客。”
“列出 1985 年 5 月 16 日聘用的教员的姓名。”
“保留球馆当前的联赛日程安排是什么样的？”
“5 号球队有哪些投球手？”

要回答这些问题，必须再次增加 SQL 词汇量，在 SELECT 语句中添加一个子句——WHERE 子句。

6.1.1　WHERE 子句

你在 SELECT 语句中使用 WHERE 子句来筛选从表中提取的数据。WHERE 子句包含一个用作筛选器的**查找条件**，这个查找条件提供了相关的机制，让你能够只选择所需的行或排除不需要的行。数据库系统根据查找条件检查 FROM 子句定义的逻辑表中的每一行。图 6-1 说明了包含 WHERE 子句的 SELECT 语句的语法。

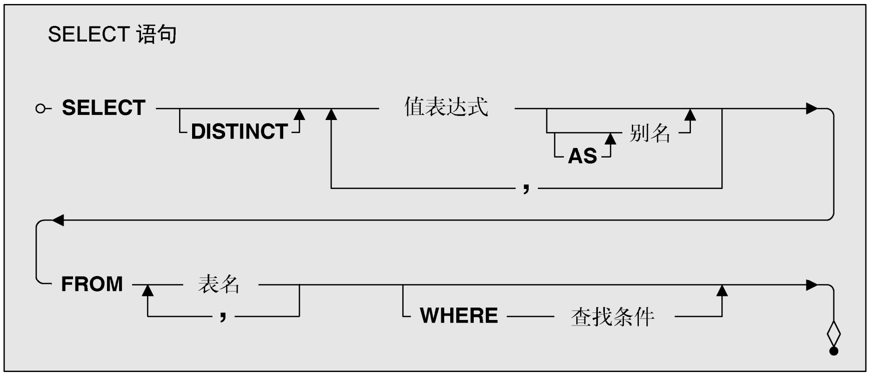


图 6-1　包含 WHERE 子句的 SELECT 语句的语法图

查找条件包含一个或多个**谓词**，其中每个谓词都是一个表达式，对一个或多个值表达式进行检查，并返回真、假或不确定。后面你将学到，可使用布尔运算符 AND 或 OR 将多个谓词合并成一个查找条件。对于特定的行，如果整个查找条件为真，这行就将出现在最终的结果集中。请注意，在查找条件只包含一个谓词的情况下，**查找条件**和**谓词**为同义词。

第 5 章说过，值表达式可包含列名、字面量值、函数或其他值表达式。编写谓词时，通常至少会包含一个这样的值表达式，即引用了 FROM 子句指定的表中的某列。

最简单也可能是最常用的谓词是将一个值表达式（列）同另一个值表达式（字面量）进行比较。例如，如果只想从 Customers 表中获取姓为 Smith 的行，可编写一个将表示姓的列同字面量值 Smith 进行比较的谓词。

```
SQL          SELECT CustLastName
             FROM Customers
             WHERE CustLastName = 'Smith'
```

上述 WHERE 子句中的谓词相当于对 Customers 表中的每行提出如下问题：这位顾客姓 Smith 吗？对于 Customers 表中的特定行，如果这个问题的答案是肯定的（真），这行将出现在结果集中。

SQL 标准定义了 18 个谓词，本章将介绍其中较为简单的 5 个：比较、BETWEEN、IN、LIKE 和 IS NULL。

- **比较**：使用 6 个比较运算符之一比较两个值表达式，这 6 个比较运算符为=（等于）、<>（不等）、<（小于）、>（大于）、<=（小于等于）、>=（大于等于）。
- **BETWEEN**（范围）：谓词 BETWEEN 让你能够检查给定值表达式的值是否在指定范围内。范围是通过使用两个由关键字 AND 分隔的值表达式指定的。
- **IN**（成员资格）：使用谓词 IN 可检查给定值表达式的值是否与指定值中的某个匹配。
- **LIKE**（模式匹配）：谓词 LIKE 让你能够检查一个字符串值表达式是否与指定的字符串模式匹配。
- **IS NULL**：使用谓词 IS NULL 可判断一个值表达式的结果是否为 Null。

❖ **说明**　对于最新的 SQL 标准中定义的其他 13 个谓词（Similar、Regex、Quantified、Exists、Unique、Normalized、Match、Overlaps、Distinct、Member、Submultiset、Set 和 Type），请不要过多地考虑。我发现，这些谓词中有 10 个没有任何商用实现。MySQL 和 PostgreSQL 支持 Regex。第 11 章将介绍另外两个谓词——Quantified 和 Exists。

6.1.2　使用 WHERE 子句

探索 SQL 标准定义的各个基本谓词前，再来看一个演示如何编写简单 WHERE 子句的示例。这次我将详细介绍完成请求的步骤。

> ❖ **说明**　贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”方法。

假设你要向数据库发出如下请求：

“居住在华盛顿州的顾客都叫什么名字？”

对这种请求进行转换时，必须尽可能明确而清晰地指出要在结果集中看到哪些信息。相比于以前，你将花更多的精力来修改请求，但这是非常值得的。下面演示了如何转换这个请求。

转换	Select first name and last name from the customers table for those customers who live in Washington State（从顾客表中选择居住在华盛顿州的顾客的名和姓）

你将像往常那样整理这个转换结果，但还将完成两个额外的任务。首先，查找表示某种约束的词语。确定的线索包括词语“何地”“谁”和“的”。下面是你要找出的一些词语类型：

- ❑ 居住在贝尔维尤的；
- ❑ 邮政编码为 98125 的；
- ❑ 5 月份下过单的；
- ❑ 加州的供应商的；
- ❑ 1985 年 5 月 16 日聘用的；
- ❑ 区号为 425 的；
- ❑ Mike Hernandez 的。

找到这样的约束条件后，便可执行第二个任务了。仔细研究这些词语，并尝试确定要检查哪列、要将该列与什么值进行比较以及如何检查该列。这些问题的答案有助于你编写 WHERE 子句中的查找条件。下面根据转换结果来回答这些问题。

- ❑ 要检查哪列？State。
- ❑ 要与什么值进行比较？'WA'。
- ❑ 如何检查这列？使用等于运算符。

你必须熟悉要用来应答请求的表的结构，如果必要，在回答这些问题前确保手边有该表结构的副本。

> ❖ **说明**　有时候这些问题的答案显而易见，而有时候它们是隐含的。在本章后面的示例中，我将演示如何拨开迷雾找到正确的答案。

回答这些问题后，根据答案编写合适的条件。接下来，将原来的约束删除，用单词 WHERE 和刚编写的查找条件取而代之。下面显示了完成这项任务后的整理结果是什么样的：

整理	Select first name and last name from ~~the~~ customers ~~table for those customers who live in~~ where state ~~is equal to~~ = 'WA' ~~Washington State~~

现在可以将整理结果转换为合适的 SELECT 语句了：

SQL

```
SELECT CustFirstName, CustLastName
FROM Customers
WHERE CustState = 'WA'
```

在这条 SELECT 语句的结果集中，将只显示那些居住在华盛顿州的顾客。

这就是定义 WHERE 子句的整个过程。本节开头说过，编写合适的查找条件并将其加入到 WHERE 子句中是一件很简单的事情，难的是定义查找条件。

6.2 定义查找条件

知道如何编写简单的 WHERE 子句后，下面深入研究一下你可定义的 5 种基本谓词。

6.2.1 比较

最常见的条件是使用比较谓词来比较两个值表达式。可使用下面的比较谓词运算符来定义 6 种比较，如图 6-2 所示。

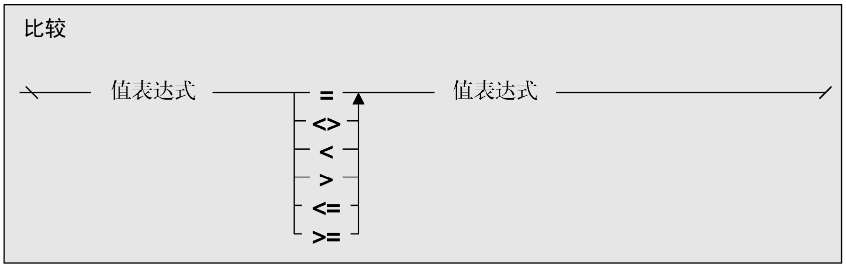


图 6-2 比较条件的语法图

=：等于	<：小于	<=：小于等于
<>：不等	>：大于	>=：大于等于

1. 字符串值比较的注意事项

比较数值数据或日期时间数据很容易，但比较字符串时必须特别小心。例如，比较两个看起来类似的字符串（如“Mike”和“MIKE”）时，结果可能出乎意料。对所有的字符串比较来说，决定因素都是数据库系统使用的排序序列。排序序列决定了如何对字符串进行排序，还将影响你如何使用其他比较条件。

鉴于很多厂商都实现了 SQL，这些实现针对体系结构不同的计算机，支持除英语外的众多其他语言，因此 SQL 标准没有指定对字符串进行排序和比较时使用的默认排序序列。字符将如何按从低到高的顺序排列取决于你使用的数据库软件，有时还取决于软件是如何安装的。

很多数据库系统使用 ASCII 排序序列，将数字放在字母前面，并将所有大写字母都放在所有小写字母前面。如果你使用的数据库系统支持 ASCII 排序序列，字符按从低到高的排列顺序将如下所示：

. . . 0123456789 . . . ABC . . . XYZ . . . abc . . . xyz . . .

然而，有些系统支持不区分大小写，如认为小写字母 a 与大写字母 A 相等。如果你的数据库系统使用的是 ASCII 排序序列，且支持不区分大小写，字符按从低到高的排列顺序将如下所示：

. . . 0123456789 . . . {Aa}{Bb}{Cc} . . . {Xx}{Yy}{Zz} . . .

请注意，位于同一对大括号（{}）内的字符被认为是相等的，因为不区分大小写。排序时不考虑大小写。

在 IBM 大型机上运行的数据库系统使用 IBM 专用的 EBCDIC 排序序列。在使用 EBCDIC 的数据库系统中，排在最前面的是小写字母，然后是大写字母，最后是数字。如果你的数据库系统使用的是 EBCDIC，字符按从低到高的排列顺序将如下所示：

. . . abc . . . xyz . . . ABC . . . XYZ . . . 0123456789 . . .

为帮助你理解这一点，我们使用一组示例列值演示排序序列对数据库系统进行比较带来的影响。

下表列出了一系列列值，它们是根据 ASCII 排序序列排列的，排列时区分大小写（数字在最前面，然后是大写字母，最后是小写字母）。

Company Name
3rd Street Warehouse
5th Avenue Market
Al's Auto Shop
Ashby's Cleaners
Zebra Printing
Zercon Productions
allegheny & associates
anderson tree farm
zorn credit services
ztech consulting

接下来，不区分大小写，将同一个字母的大写版本和小写版本视为相等，结果如下表所示。

Company Name
3rd Street Warehouse
5th Avenue Market
Al's Auto Shop
allegheny & associates
anderson tree farm
Ashby's Cleaners
Zebra Printing
Zercon Productions
zorn credit services
ztech consulting

最后，来看看在使用 EBCDIC 排序序列的 IBM 系统上，这些值是如何排列的（依次为小写字母、大写字母和数字）。

Company Name
allegheny & associates
anderson tree farm
zorn credit services
ztech consulting
Al's Auto Shop
Ashby's Cleaners
Zebra Printing
Zercon Productions
3rd Street Warehouse
5th Avenue Market

比较两个长度不同的字符串（如"John"和"John"或"Mitch"和"Mitchell"）时，结果也可能出乎意料。所幸 SQL 标准明确地规定了数据库系统必须如何处理这一点。对两个长度不同的字符串进行比较前，数据库系统必须在较短的字符串右边添加**默认的填充字符**，直到其长度与较长的字符串相同（在大多数数据库系统中，默认的填充字符为空格），再根据排序序列判断这两个字符串是否相等。因此，（经过填充后）"John"和"John"相等，而"Mitch"和"Mitchell"不相等。

> ❖ **说明**　有些数据库系统采取不同于 SQL 标准的做法，它们忽略末尾的空格，而不是用空格填充较短的字符串。因此，在这些系统中，“John”和“John”也被视为相等，但原因不同——忽略了第二个字符串末尾的空格。务必对你使用的数据库系统进行检查，确定它是如何处理这种类型的比较的，还有它返回的结果是否出乎意料。

总之，务必查看你使用的数据库系统的文档，确定它是如何对大写字母、小写字母和数字进行排序的。

2. 相等和不等

虽然你已经见过几个相等比较示例，我们还是再来看看使用相等运算符的相等比较条件。

假设你要向数据库发出如下请求：

“显示 1977 年 3 月 14 日聘用的所有经纪人的名和姓。”

由于要查找特定的聘用日期，因此可使用相等运算符编写一个相等比较条件来检索合适的信息。下面执行转换过程，以编写合适的 SELECT 语句。

转换	Select first name and last name from the agents table for all agents hired on March 14, 1977（从经纪人表中选择所有 1977 年 3 月 14 日聘用的经纪人的名和姓）
整理	Select first name ~~and~~ last name from ~~the~~ agents ~~table for all agents hired on~~ where date hired = ~~March 14, 1977~~ ‘1977-03-14’

SQL	`SELECT AgtFirstName, AgtLastName`
	`FROM Agents`
	`WHERE DateHired = '1977-03-14'`

在这个示例中，我检查特定列，确定其中是否有与指定日期值匹配的值。大致而言，我执行了一个**包含**过程：对于 Agents 表中的行，如果其 DateHired 列的值与指定的日期匹配，它就将包含在结果集中。但如果你要做相反的操作，将某些行**排除**在结果集之外，该怎么办呢？在这种情况下，可编写一个使用不等运算符的比较条件。

假设你要提交如下请求：

“给我一个清单，列出除位于贝尔维尤外的所有供应商的名称和电话号码。”

你可能已确定需要将位于贝尔维尤的供应商排除在外，因此可使用不等条件来完成这项任务。词语“除……外”清楚地表明使用不等条件是合适的。转换时请牢记这一点。

转换	Select vendor name and phone number from the vendors table for all vendors except those based in ‘Bellevue’（从供应商表中选择所有供应商的名称和电话号码，但排除位于贝尔维尤的供应商）
整理	Select vendor name ~~and~~ phone number from ~~the~~ vendors ~~table for all vendors except those based in~~ where city <> 'Bellevue'

SQL	`SELECT VendName, VendPhone`
	`FROM Vendors`
	`WHERE VendCity <> 'Bellevue'`

> ❖ **说明**　SQL 标准将符号<>用作不等运算符。多个 RDBMS 程序都提供了不同的表示法，如!=（Microsoft SQL Server 和 Sybase）和 ¬=（IBM DB2）。务必查看你使用的数据库系统的文档，确定它使用哪种方式来表示不等运算符。

使用这个简单条件，便将位于贝尔维尤的所有供应商排除在外了。本章后面将介绍另一种从结果集中排除行的方法。

3. 小于和大于

经常需要返回某列的值小于或大于特定值的行，这种比较是使用比较运算符<（小于）、<=（小于等于）、>（大于）或>=（大于等于）来实现的。比较的值的数据类型决定了它们的关系。

- **字符串**：这种比较判断第一个值表达式的值是在第二个表达式的值的前面（<）还是后面（>）。例如，可将 a < c 解读为“a 在 c 的前面吗”？有关排序序列的详情，请参阅前面的“字符串值比较的注意事项”。

- **数字**：这种比较判断第一个值表达式的值是否小于（<）或大于（>）第二个值表达式的值。例如，可将 10 > 5 解读为“10 大于 5 吗”？
- **日期/时间**：这种比较判断第一个值表达式的值是否早于（<）或晚于（>）第二个值表达式的值。例如，可将‘2007-05-16’<‘2007-12-15’解读为“2007 年 5 月 16 日早于 2007 年 12 月 15 日吗”。日期和时间是根据时间顺序进行比较的。

我们来看看如何使用这些比较谓词来应答请求。

“是否存在发货日期早于下单日期的订单？”

在这个示例中，将使用比较运算符<，因为要判断的是发货日期是否早于下单日期。下面演示了如何转换这个请求。

转换	Select order number from the orders table where the ship date is earlier than the order date（从订单表中选择发货日期早于下单日期的订单号）
整理	Select order number from ~~the~~ orders ~~table~~ where ~~the~~ ship date ~~is earlier than the~~ < order date
SQL	SELECT OrderNumber FROM Orders WHERE ShipDate < OrderDate

在这条 SELECT 语句的结果集中，只包含 Orders 表中满足查找条件的行。

下面的示例要求使用比较运算符>来检索合适的信息。

“是否有超过 4 学分的课程？”

转换	Select class ID from the classes table for all classes that earn more than four credits（从课程表中选择学分超过 4 的课程 ID）
整理	Select class ID from ~~the~~ classes ~~table for all classes that earn more than four~~ where credits > 4
SQL	SELECT ClassID FROM Classes WHERE Credits > 4

在这条 SELECT 语句生成的结果集中，只包含学分为 5 或更多的课程，如 Intermediate Algebra（中级代数）和 Engineering Physics（工程物理）。

下面来看一些这样的示例，即你不仅对大于或小于目标值的值感兴趣，还对等于目标值的值感兴趣。

“给我提供从 1989 年 1 月 1 日起聘用的所有员工的姓名。”

这里将使用大于等于比较，因为要检索的是从 1989 年 1 月 1 日（含）起到当前的所有聘用日期。转换时务必确定要在 SELECT 子句中包含的所有列。

<table>
<tr><td>转换</td><td>Select first name and last name as EmployeeName from the employees table for all employees hired since January 1, 1989（从员工表中选择从 1989 年 1 月 1 日起聘用的所有员工的名和姓，并将名和姓拼接为 EmployeeName）</td></tr>
<tr><td>整理</td><td>Select first name ~~and~~ || ' ' || last name as EmployeeName from ~~the~~ employees ~~table for all employees hired since~~ where date hired >= ~~January 1, 1989~~ ‘1989-01-01’</td></tr>
<tr><td>SQL</td><td>SELECT FirstName || ' ' || LastName
    AS EmployeeName
FROM Employees
WHERE DateHired >= '1989-01-01'</td></tr>
</table>

下面是另一个你可能向数据库发出的请求：

“给我一个清单，列出零售价不超过 50 美元的商品。”

你可能推断出了，这个请求需要使用小于等于比较。这确保 SELECT 语句的结果集只包含价格为 1 美分到 50 美元的商品。下面演示了如何转换这个请求。

转换	Select product name from the products table for all products with a retail price of fifty dollars or less（从商品表中选择零售价不超过 50 美元的商品名称）
整理	Select product name from ~~the~~ products ~~table for all products with a~~ where retail price of <= 50 ~~fifty dollars or less~~

SQL	`SELECT ProductName`
	`FROM Products`
	`WHERE RetailPrice <= 50`

前面的每个示例都只使用了一种比较。本章后面将演示如何使用 AND 和 OR 来合并比较。

6.2.2 范围

使用范围条件可检查值表达式的值是否在特定范围内。图 6-3 说明了这种条件的语法。

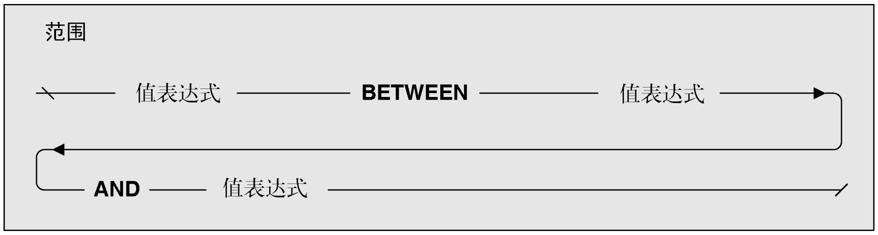


图 6-3 范围条件的语法图

范围条件检查给定值表达式的值是否在两个值表达式定义的范围内。谓词 BETWEEN . . . AND 使用两个值表达式定义来范围，其中第一个值表达式为范围的起点，第二个为范围的终点；起点和终点都包含在范围内。仅当给定值表达式的值在指定范围内时，相应的行才会出现在结果集中。

使用 BETWEEN . . . AND 时有一点需要注意。SQL 标准实际上定义了两种 BETWEEN 比较：对称的和非对称的。默认是非对称的，即 Value1 BETWEEN Value2 AND Value3 等价于 Value1 >= Value2 AND Value1 <= Value3。这意味着 Value2 必须小于等于 Value3，这个谓词才能起作用。例如，SQL 标准规定，对于 `MyColumn BETWEEN 5 AND 10`，应将其解读为 `MyColumn >= 5 AND MyColumn <= 10`，因此如果将较大的值放在前面，如 `MyColumn BETWEEN 10 AND 5`，将解读为 `MyColumn >=10 AND MyColumn <= 5`，而这永远不可能为真！（一个列值不可能在大于等于 10 的同时小于等于 5。）然而，有些数据库系统允许 Value2 大于等于 Value3，这与 SQL 标准中使用关键字 SYMMETRIC 等效（据我所知，还没有任何主要实现支持关键字 ASYMMETRIC 和 SYMMETRIC）。有关这方面的细节，请参阅你使用的数据库系统的文档。

下面通过几个示例演示如何使用范围条件。

> “哪些教员是 1986 年 7 月聘用的？”

这里适合使用范围条件，因为要检索的是在一组特定日期（这里是 1986 年 7 月 1 日到 31 日）聘用的所有员工的姓名。下面来执行转换并编写合适的 SELECT 语句。

转换	Select first name and last name from the staff table where the date hired is between July 1, 1986, and July 31, 1986（从教员表中选择聘用日期在 1986 年 7 月 1 日到 7 月 31 日之间的教员的名和姓）
整理	Select first name and last name from ~~the~~ staff ~~table~~ where ~~the~~ date hired is between ~~July 1, 1986~~ '1986-07-01' and ~~July 31, 1986~~ '1986-07-31'

SQL	`SELECT FirstName, LastName`
	`FROM Staff`
	`WHERE DateHired`
	`BETWEEN '1986-07-01' AND '1986-07-31'`

请注意，相比于请求，转换结果指定的日期更明确。应以尽可能清晰的方式对请求进行转换，这样才能编写出合

适的 SELECT 语句。

也可将范围条件用于字符串数据，如下面的示例所示：

“给我一个清单，列出姓以字母 B 打头的学生的姓名和电话号码。”

转换	Select last name, first name, and phone number from the students table for all students whose last name begins with the letter ‘B’（从学生表中选择姓以字母 B 打头的学生的姓、名和电话号码）
整理	Select last name, first name, ~~and~~ phone number from ~~the~~ students ~~table for all students whose name begins with the letter ‘B’~~ where last name between 'B' and 'Bz'

SQL	`SELECT StudLastName, StudFirstName,`
	`    StudPhoneNumber`
	`FROM Students`
	`WHERE StudLastName BETWEEN 'B' AND 'Bz'`

编写与字符串数据相关的范围时，请仔细想想你要包含的值。例如，下面是在这个请求中指定范围起点和终点的三种方式，它们得到的结果截然不同！

- BETWEEN ‘A’ AND ‘B’：我知道，很多读者不会将 A 作为起点，因为他们知道使用这个范围将包含所有姓以 A 打头的学生，但这是一种常见的错误。
- BETWEEN ‘B’ AND ‘C’：就这个示例而言，这样指定起点和终点也许能够返回所需的结果，但根据要比较的字符数据，这可能返回意外的结果。别忘了，BETWEEN 运算符指定的范围包含起点和终点，因此如果有学生的姓为 C，他将包含在结果集中。
- BETWEEN ‘B’ AND ‘BZ’：这是最清晰、最明确的起点和终点指定方法——在大多数情况下将返回所需的结果。归根结底，要定义正确的范围，你必须熟悉目标数据。

结束对 BETWEEN 的讨论前，我们再说一点。图 6-3 指出，不仅 BETWEEN 子句中的两个值可以是**值表达式**，第一个值也可以是。前面说过，值表达式可以很简单，如列名或字面量，也可以很复杂，如字符表达式、数学表达式或日期时间表达式。如果表中有两个定义范围的列（如示例数据库 Entertainment Agency 中 Engagements 表中的 StartDate 和 EndDate 列），也可使用 BETWEEN 来查找这样的行，即指定的值位于该行中这两列的值之间。下面是一个这样的示例。

“列出所有在 2017 年 10 月 10 日进行的演出。”

转换	Select engagement number, start date, and end date from the engagements table for engagements where October 10, 2017, is between the start date and the end date（从演出合约表中选择 2017 年 10 月 10 日位于其开始日期和结束日期之间的演出的演出合约的编号、开始日期和结束日期）
整理	Select engagement number, start date, ~~and~~ end date from ~~the~~ engagements ~~table for engagements~~ where ~~October 10, 2017 is~~ '2017-10-10' between ~~the~~ start date and ~~the~~ end date

SQL	`SELECT EngagementNumber, StartDate, EndDate`
	`FROM Engagements`
	`WHERE '2017-10-10' BETWEEN StartDate AND EndDate`

至此，我演示了如何使用笼统和具体的值范围来缩小请求范围。下面来看看如何使用具体的值列表来缩小请求范围。

6.2.3　集成员资格

要检查一个值表达式的值是否是一系列值中的一个，可使用成员资格条件。如图 6-4 所示，成员资格条件使用谓词 IN 来判断第一个值表达式的值是否是一系列值中的一个，而这一系列值是在括号中使用一个或多个值表达式定义的。

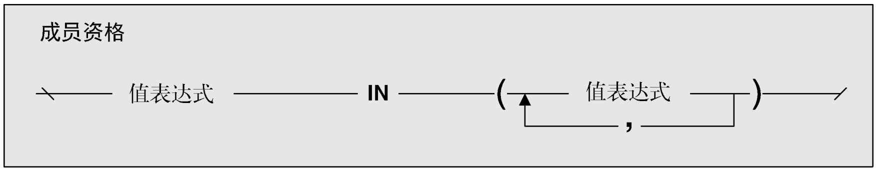


图 6-4　成员资格条件的语法图

从理论上说，可在列表中包含的值表达式数量几乎不受限制，但比较合理的做法是只包含几个，因为要表示较大的值范围，可使用前面介绍的两种条件。在只需指定有限的值列表时，使用成员资格条件的效果最佳，如下面的示例所示。

下面是一个你可能向数据库发出的请求：

“我需要知道 2017 年如下日期的联赛在哪些保龄球馆举行：9 月 18 日、10 月 9 日和 11 月 6 日。”

这种请求适合使用成员资格条件来应答，因为它要查找一系列特定的值。如果请求没这么明确，你很可能使用范围条件。下面演示了如何转换这个请求。

转换	Select tourney location from the tournaments table where the tourney date is in this list of dates: September 18, 2017; October 9, 2017; November 6, 2017（从联赛表中选择比赛日期为 2017 年 9 月 18 日、2017 年 10 月 9 日或 2017 年 11 月 6 日的联赛的比赛地点）
整理	Select tourney location from ~~the~~ tournaments ~~table~~ where ~~the~~ tourney date is in ~~this list of dates~~: (~~September 18, 2017~~; '2017-09-18',~~October 9 2017~~; '2017-10-09', ~~November 6, 2017~~ '2017-11-06')

SQL	`SELECT TourneyLocation`
	`FROM Tournaments`
	`WHERE TourneyDate`
	`   IN ('2017-09-18', '2017-10-09',`
	`   '2017-11-06')`

再来看一个需要使用成员资格条件来应答的请求：

“我们在西雅图、雷德蒙德和博塞尔有哪些演唱组合？”

转换	Select stage name from the entertainers table for all entertainers based in Seattle, Redmond, or Bothell（从演唱组合表中选择所有位于西雅图、雷德蒙德或博塞尔的演唱组合的艺名）
整理	Select stage name from the entertainers ~~table for all entertainers based~~ where city in (‘Seattle’, ‘Redmond’, ~~or~~ ‘Bothell’)

SQL	`SELECT EntStageName`
	`FROM Entertainers`
	`WHERE EntCity`
	`   IN ('Seattle', 'Redmond', 'Bothell')`

你可能注意到了，在转换结果中的城市列表中，我使用了词语“或”而不是请求中的“和”。这样做的原因和逻辑很简单：在给定演唱组合的 EntCity 列中，只有一个城市。在特定的行中，不可能同时包含西雅图、雷德蒙德和博塞尔，但可能包含西雅图、雷德蒙德或博塞尔。这一点好像无关紧要，但使用合适的词语可让转换和整理结果更清晰，确保为请求编写的 SELECT 语句最合适。等你到本章后面开始使用多个条件时将发现，这个看似微不足道的地方将变得更为重要。

前面介绍的所有条件都将整个值作为判断依据，下面介绍一种让你能够将部分值作为判断依据的条件。

6.2.4　模式匹配

需要找出类似于给定模式字符串的值或要将部分信息作为查找条件时，模式匹配条件很有用。这种条件的语法如图 6-5 所示。

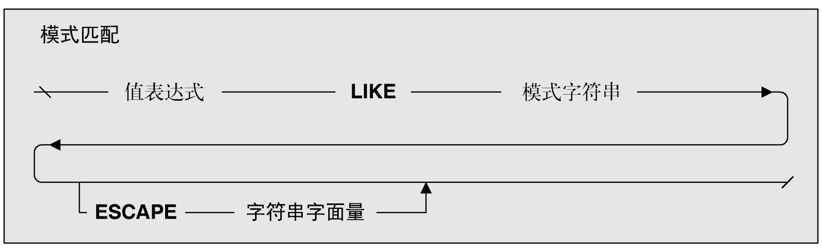


图 6-5　模式匹配条件的语法图

这种条件使用谓词 LIKE 检查值表达式的值是否与指定的模式字符串匹配。模式字符串可包含任何常规字符和两个特殊的通配字符：百分比符号（%）和下划线（_）。百分比字符表示任意数量（包括零个）的常规字符，而下划线表示单个常规字符。你指定的模式字符串决定了将获取哪些值。表 6-1 列出了可定义的各种模式字符串。

> ❖ **说明**　Microsoft Office Access 是最流行的数据库系统之一，它使用星号（*）和问号（?），而不是百分比符号（%）和下划线（_）。Access 还支持使用井号（#）在特定位置查找数字字符。如果你使用的是 Microsoft Access，请在 LIKE 谓词的模式字符串中替换这些字符。

表 6-1　模式字符串示例

模式字符串	准　　则	返回值示例
`'Sha%'`	以“Sha”打头的任意长度的字符串	**Sha**nnon、**Sha**ron、**Sha**wn
`'%son'`	以“son”结尾的任意长度的字符串	Ben**son**、John**son**、Morri**son**
`'%han%'`	包含“han”的任意长度的字符串	Buc**han**an、**han**del、Jo**han**sen、Nat**han**son
`'Ro_'`	以“Ro”打头且包含 3 个字符的字符串	**Ro**b、**Ro**n、**Ro**y
`'_im'`	以“im”结尾且包含 3 个字符的字符串	J**im**、K**im**、T**im**
`'_ar_'`	包含 4 个字符且第 2 个和第 3 个字符为“ar”的字符串	B**ar**t、G**ar**y、M**ar**k
`'_at%'`	第 2 个和第 3 个字符为“at”的任何字符串	G**at**es、M**at**thews、P**at**terson
`'%ac_'`	倒数第 3 个和第 2 个字符为“ac”的任何字符串	Apod**ac**a、Tr**ac**y、Wall**ac**e

我们来看看如何使用模式匹配条件。请看下面的请求：

“给我一个清单，列出姓以 Mar 打头的顾客。”

类似这样的请求通常包含的词语指出了需要使用模式匹配条件。下面是一些你可能会遇到的词语：

- 以 Her 打头；
- 开头为 Ba；
- 包含单词 Park；
- 包含字母 han；
- 中间有 ave；
- 末尾为 son；
- 以 ez 结尾。

> ❖ **警告**　在很多数据库系统中，字符串比较是区分大小写的。多个主要数据库系统都允许系统管理员在安装数据库服务器时设置一个选项，以指定比较字符串时区分还是不区分大小写。如果你使用的数据库系统是区分大小写的，`LIKE '%chi%'`将与“roast chicken”匹配，但与“Chicken a la King”不匹配，因为模式字符串中的“c”与列值中的“C”不相等。请查看你使用的数据库系统的文档，确定是否需要区分大写字母和小写字母。

如你所见，确定需要使用什么样的模式字符串来应答请求比较容易。知道需要编写什么样的模式字符串后，就可以继续完成转换过程了。

转换	Select last name and first name from the customers table where the last name begins with ‘Mar’（从顾客表中选择姓以 Mar 打头的顾客的姓和名）
整理	Select last name ~~and~~ first name from ~~the~~ customers ~~table~~ where ~~the~~ last name ~~begins with~~ like ‘Mar%’

SQL	`SELECT CustLastName, CustFirstName`
	`FROM Customers`
	`WHERE CustLastName LIKE 'Mar%'`

在这条 SELECT 语句的结果集中，包含姓 Marks、Marshall、Martinez、Marx 等顾客的姓名，因为它只关心姓的前三个字符是否匹配。

再看一个使用模式匹配条件来应答请求的例子：

“给我一个清单，列出街道地址包含单词‘Forest’的供应商的名称。”

转换	Select vendor name from the vendors table where the street address contains the word ‘Forest’（从供应商表中选择街道地址包含单词 Forest 的供应商的名称）
整理	Select vendor name from ~~the~~ vendors ~~table~~ where ~~the~~ street address ~~contains the word~~ like ‘%Forest%’

SQL	`SELECT VendName`
	`FROM Vendors`
	`WHERE VendStreetAddress LIKE '%Forest%'`

在这里，对于 Vendors 表中的行，仅当其街道地址包含诸如 Forest Park Place、Forest Ridge Avenue、Evergreen Forest Drive、Black Forest Road 等街道名时，才会出现在结果集中。

虽然使用合适的通配字符可查找任何模式字符串，但如果要检索的值包含百分比符号或下划线，将出现麻烦。例如，如果你要检索值 MX_445，将出现麻烦，因为它包含下划线。要消除这种可能的困境，可在 LIKE 谓词中使用选项 ESCAPE，如图 6-5 所示。

选项 ESCAPE 让你能够指定只包含**一个**字符的字符串字面量（这被称为**转义字符**），从而告诉数据库系统该如何解读模式字符串中的百分比符号和下划线。要指定转义字符，可将其放在关键字 ESCAPE 的后面，并像其他字符串字面量一样使用单引号将其括起。在模式字符串中，如果通配字符前面是转义字符，数据库系统将把它解读为实际字符。

下面的示例演示了如何使用选项 ESCAPE：

“给我一个清单，列出代码由‘G_00’和一个字符组成的商品。”

转换	Select product name and product code from the products table where the product code begins with ‘G_00’ and ends in a single number or letter（从商品表中选择代码由 G_00 和一个字符组成的商品的名称和代码）
整理	Select product name ~~and~~ product code from ~~the~~ products ~~table~~ where ~~the~~ product code ~~begins with~~ like 'G_00_' ~~and ends in a single number or letter~~

SQL	`SELECT ProductName, ProductCode`
	`FROM Products`
	`WHERE ProductCode LIKE 'G\_00_' ESCAPE '\'`

要应答这个请求，显然需要使用选项 ESCAPE，否则数据库系统将把模式字符串中的下划线解读为通配字符。注意，这里在整理结果中包含了转义字符。你在整理结果中也应这样做，因为这将确保你在定义 SELECT 语句时使用选项 ESCAPE。

这条 SELECT 语句将返回诸如 G_002、G_00X 等商品代码。由于这里要查找的是标准中定义的两个通配字符之一，因此必须包含选项 ESCAPE。如果我使用 LIKE 'G_00_'，数据库系统将返回商品代码为下面这样的所有行：第一个字符为‘G’，第 2 个字符为任何字符（因为这里使用了通配字符_），第 3 个和第 4 个字符为 0，第 5 个字符为任何字符。将“\”指定为转义字符后，数据库系统将忽略这个转义字符，但将第一个下划线解读为实际字符而不是通配字符。对于第二个下划线，由于它前面没有转义字符，因此数据库系统将它把解读为通配字符。

请别忘了，不能将要检索的值中包含的字符指定为转义字符。如果你要查找诸如 Martin & Lewis、Smith & Kearns、Hernandez & Viescas 等值，就不能将“&”用作转义字符。另外别忘了，转义字符只影响紧跟在它后面的通配字符，但在模式字符串中可使用任意数量的转义字符。

6.2.5 Null

前面介绍了如何查找整个值和值的一部分，下面来讨论如何查找未知值。第 5 章说过，Null 表示的并不是零、只包含空格的字符串或长度为零的字符串（不包含任何字符的字符串），因为这些内容在特定情况下都是有意义的。相

反，它表示值缺失或未知。要检查值表达式是否为 Null 值，可使用如图 6-6 所示的 Null 条件。

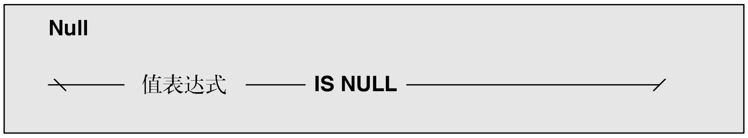


图 6-6　Null 条件的语法图

这种条件使用谓词 IS NULL 判断值表达式的值是否为 Null，是一种非常直观的操作。下面来看看如何使用这种条件，如下面的示例所示：

“给我一个清单，列出没有指出居住在哪个县的顾客。”

转换	Select first name and last name as Customer from the customers table where the county name is unspecified（从顾客表中选择未指定县名的顾客的名和姓，并将它们拼接为 Customer）
整理	Select first name || ‘ ’ || ~~and~~ last name as Customer from ~~the~~ customers ~~table~~ where ~~the~~ county name is null ~~unspecified~~
SQL	SELECT CustFirstName || ' ' || CustLastName AS Customer FROM Customers WHERE CustCounty IS NULL

在这条 SELECT 语句的结果集中，只包含不知道或不记得自己居住在哪个县，或者居住在华盛顿特区的顾客（顺便说一句，在整个美国，华盛顿特区是唯一一座不属于任何县的城市）。

再来看一个你可能向数据库发出的请求：

“哪些演出合约还没有确定合约价格？”

转换	Select engagement number and contract price from the engagements table for any engagement that does not have a contract price（从演出合约表中选择没有合约价格的演出合约的编号和合约价格）
整理	Select engagement number ~~and~~ contract price from ~~the~~ engagements ~~table for any engagement that does not have a~~ where contract price is null
SQL	SELECT EngagementNumber, ContractPrice FROM Engagements WHERE ContractPrice IS NULL

从表面上看，这好像是一个很简单的请求，只需查找所有合约价格为零的演出合约即可。但外表可能迷惑你，让你做出错误的假设。在这个例子中，如果演出机构将促销演出的合约价格定为零，那么零就是合法而有意义的值。因此，还未确定或商谈好的合约价格其实应该为 Null。

这个示例表明，要向数据库发出准确而有意义的请求，必须了解目标数据。执行 SELECT 语句后，如果你认为结果集中的信息是错误的，不要难过。你首先想到的可能是重写整条 SELECT 语句，因为你认为自己犯了严重的语法错误。但采取任何大动作前，请重新审视目标数据，确保自己清楚地知道你当前是如何使用它们的。对数据有更深入的认识后，你通常会发现，只需对 SELECT 语句做细微的修改，就能检索到正确的信息。

❖ **说明**　要在值表达式中查找 Null 值，必须使用 Null 条件。诸如<值表达式> = Null 这样的条件是非法的，因为根据定义，不能将值表达式的值同未知值进行比较。事实上，在任何比较谓词中，使用 Null 的结果都是未知的，而未知被视为假，因此比较的结果是不满足条件。

6.2.6　使用 NOT 排除行

至此，我演示了如何在结果集中**包含**特定的行，下面来看看如何使用 NOT 运算符将特定的行**排除**在结果集之外。

前面演示了一种将特定行排除在结果集之外的简单方式，那就是使用不同运算符来指定一个比较条件。通过使用 NOT 运算符，也可根据其他条件来排除行。如图 6-7 所示，这个运算符是谓词 BETWEEN、IN、LIKE 和 IS NULL 中一个可选的部分。在这些谓词中包含 NOT 运算符时，SELECT 语句将把满足指定条件的行排除在外，而结果集将包含那些**不满足**条件的行。

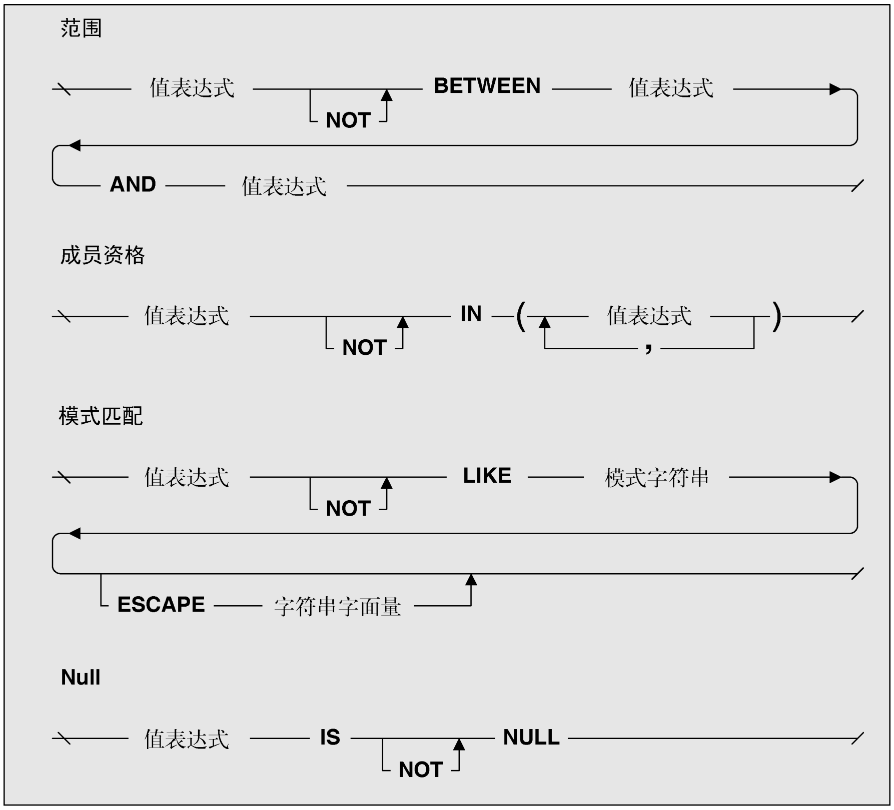


图 6-7　NOT 运算符的语法图

下面的示例演示了如何在查找条件中使用 NOT：

“给我一个清单，列出所有的订单，但 7 月下的订单除外。”

像这样的请求要求你定义一条 SELECT 语句来排除满足特定条件的行，它通常包含指出要在查找条件中包含 NOT 运算符的词语。你将遇到的这种词语类似于下面这样：

- 不以“Her”打头；
- 不属于管理部（Administrative）和人事部（Personnel）；
- 有传真号的；
- 6 月 1 日之前或 8 月 31 日之后聘用的。

有时候，为了正确地转换词语，必须做一点推理。有些词语（如前面列出的第三个）并未显式地指出需要使用 NOT 运算符。在这里，需求是隐含的，因为你要排除所有没有传真号的。着手处理包含这些类型的词语的请求时，通常需要仔细分析它们，还可能需要重写，这样才能确定是否需要将特定行排除在结果集之外。这里没有简单的经验规则，但经过一定的训练后，只要有一点点耐心，就能轻松地确定是否需要使用 NOT 运算符。

确定需要将特定信息排除在结果集之外后，就可接着完成转换过程了。

“给我一个清单，列出所有的订单，但 10 月份下的订单除外。”

转换	Select order ID and order date from the orders table where the order date does not fall between October 1, 2017, and October 31, 2017（从订单表中选择下单日期不在 2017 年 10 月 1 日和 10 月 30 日之间的所有订单的 ID 和下单日期）
整理	Select order ID ~~and~~ order date from ~~the~~ orders ~~table~~ where ~~the~~ order date ~~does~~ not ~~fall~~ between ~~October 1, 2017~~, '2017-10-01' and ~~October 31, 2017~~ '2017-10-31'

SQL	`SELECT OrderID, OrderDate`
	`FROM Orders`
	`WHERE OrderDate NOT BETWEEN '2017-10-01'`
	`   AND '2017-10-31'`

在这条 SELECT 语句生成的结果集中，不包含在 2017 年 10 月 1 日到 30 日下的订单，但将包含其他所有订单。可使用多个条件进一步限制结果集，使其只包含 2017 年下的订单，这是下一节将讨论的主题。

现在假设你要处理如下请求：

“给我一个清单，列出所有不是教授和副教授的教员的 ID。”

转换	Select staff ID and title from the faculty table where the title is not ‘professor’ or ‘associate professor’（从全体教员表中选择所有职称不是教授和副教授的教员的 ID 和职称）
整理	Select staff ID ~~and~~ title from ~~the~~ faculty ~~table~~ where ~~the~~ title ~~is~~ not in ('professor', ~~or~~ 'associate professor')

SQL	`SELECT StaffID, Title`
	`FROM Faculty`
	`WHERE Title`
	`NOT IN ('Professor', 'Associate Professor')`

在这里，需要将职称为请求中指定的职称之一的教员都排除在外，因此使用一个包含 NOT 运算符的成员资格条件，将正确的行放到结果集中。

习惯了根据具体情况分析并重写请求后，从结果集中排除特定行将变得比较容易。如你所见，真正的关键在于，能够根据给定的请求确定该使用哪种条件。

6.3　使用多个条件

前面处理的请求都很简单，只需使用一个条件就能做出应答。下面来看看如何使用多个条件来应答复杂的请求。首先来看看下面的请求：

“给我一个清单，列出居住在西雅图且姓以字母 H 打头的顾客的名和姓。”

凭借已掌握的知识，你确定要应答这个请求，需要使用一个相等比较条件和一个模式匹配条件。你确定了需要使用的条件，但如何将它们合并成一个查找条件呢？答案可在 SQL 标准定义的查找条件语法中找到，如图 6-8 所示。

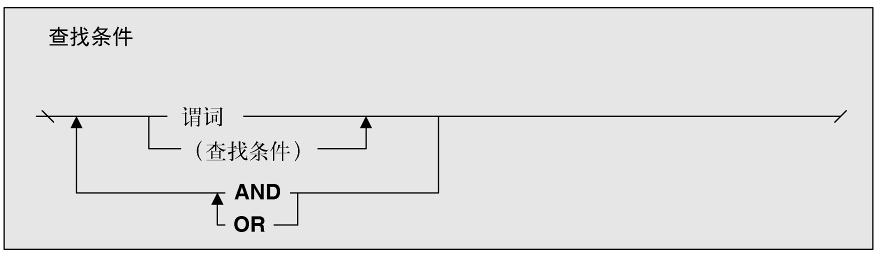


图 6-8　查找条件的语法图

6.3.1　AND 和 OR 简介

要合并多个条件，可使用运算符 AND 和 OR；为应答请求而合并的一组条件组成了单个查找条件。如图 6-8 所示，还可将合并得到的查找条件与其他条件合并，为此可将其放在括号内。这让你能够编写非常复杂的 WHERE 子句，精

确地控制结果集将包含哪些行。

1. 使用 AND

合并多个条件的第一种方式是使用 AND 运算符。仅当给定行必须满足所有条件才出现在结果集中时，就可使用这个运算符。下面以本节开头所说的请求为例，演示如何在转换过程中使用这个运算符。

“给我一个清单，列出居住在西雅图且姓以字母 H 打头的顾客的名和姓。”

转换	Select first name and last name from the customers table where the city is ‘Seattle’ and the last name begins with ‘H’（从顾客表中选择城市为 Seattle 且姓以 H 打头的顾客的名和姓）
整理	Select first name ~~and~~ last name from ~~the~~ customers ~~table~~ where ~~the~~ city ~~is~~ = 'Seattle' and ~~the~~ last name ~~begins with~~ like ‘H%’

SQL	`SELECT CustFirstName, CustLastName`
	`FROM Customers`
	`WHERE CustCity = 'Seattle'`
	`    AND CustLastName LIKE 'H%'`

你确定这个请求需要使用相等条件和模式匹配条件，并使用 AND 运算符确保它们都得到满足。不满足其中任何一个条件的行都将排除在结果集之外。

可根据要处理的请求将任意数量的条件串接起来，但别忘了，给定行要进入结果集，必须满足使用 AND 合并的所有条件。请记住，必须满足整个查找条件，给定行才会出现在结果集中。图 6-9 显示了使用 AND 运算符合并两个谓词表达式的结果。只要有一个表达式为假，相应的行就会被拒绝。

第二个表达式

第一个表达式

AND	真	假
真	真 （行被选择）	假 （行被拒绝）
假	假 （行被拒绝）	假 （行被拒绝）

图 6-9　使用 AND 运算符合并两个谓词表达式的结果

2. 使用 OR

合并多个条件的第二种方式是使用 OR 运算符。在只要有一个被合并的条件满足，给定行就将包含在结果集中时，就可使用这个运算符。下面的示例演示了如何在查找条件中使用 OR 运算符：

“给我一个清单，列出居住在西雅图或来自俄勒冈州的教员的姓名以及所在城市和州。”

转换	Select first name, last name, city, and state from the staff table where the city is ‘Seattle’ or the state is ‘OR’（从教员表中选择城市为 Seattle 或州为 OR 的教员的名、姓、城市和州）
整理	Select first name, last name, city, ~~and~~ state from ~~the~~ staff ~~table~~ where ~~the~~ city ~~is~~ = 'Seattle' or ~~the~~ state ~~is~~ = 'OR'

SQL	`SELECT StfFirstName, StfLastName, StfCity, StfState`
	`FROM Staff`
	`WHERE StfCity = 'Seattle' OR StfState = 'OR'`

在这里，你确定为应答这个请求，需要使用两个相等比较条件，并使用 OR 运算符确保只要满足其中一个条件。只要满足其中一个条件，给定行就将包含在结果集中。为帮助澄清这一点，图 6-10 列出了使用 OR 运算符合并两个谓词表达式的结果。

第二个表达式

第一个表达式

OR	真	假
真	真 （行被选择）	真 （行被选择）
假	真 （行被选择）	假 （行被拒绝）

图 6-10　使用 OR 运算符合并两个谓词表达式的结果

要确定是否该使用 AND 运算符来合并条件，相对比较简单而直观，但要确定是否该使用 OR 运算符，有时比较棘手。例如，请看下面的请求：

“给我一个清单，列出位于华盛顿州和加州的供应商的名称和电话号码。”

你首先想到的可能是使用 AND 运算符，因为条件看起来很明显：需要获取位于华盛顿州和加州的供应商。遗憾的是，你错了。只要想一想，你就会发现，一家供应商要么位于华盛顿，要么位于加州，因为对于一家供应商，只能指定一个州。现在，真正的条件是不是明显得多？本章前面说过，你必须养成对较复杂的请求进行研究和分析的习惯，尽最大的努力搞明白其中隐含的条件。

我们接着往下走，对这个请求进行转换。

“给我一个清单，列出位于华盛顿州和加州的供应商的名称和电话号码。”

转换	Select name, phone number, and state from the vendors table where the state is ‘WA’ or ‘CA’（从供应商表中选择州为 WA 或 CA 的供应商的名称、电话号码和州）
整理	Select name, phone number, ~~and~~ state from ~~the~~ vendors ~~table~~ where ~~the~~ state ~~is~~ = 'WA' or state = 'CA'

SQL	`SELECT VendName, VendPhoneNumber, VendState`
	`FROM Vendors`
	`WHERE VendState = 'WA' OR VendState = 'CA'`

你考虑了两个相等比较条件，并使用 OR 运算符确保必须满足其中一个条件。然而，注意，在整理结果和 SQL 语句的查找条件中，“州”出现了两次，这是必须的，因为每个比较条件都遵循如下语法：

值表达式 <比较运算符> 值表达式

别忘了，除非子句、关键字或术语被明确规定是可选的，否则在语法中不能省略它们，因此像 `WHERE VendState = 'WA' OR 'CA'`这样的条件是绝对非法的。你可能会问：为何会这样呢？本章后面将更深入地介绍表达式运算符的执行顺序——优先级。

在这个示例中，数据库系统将严格按照从左到右的顺序计算表达式，因此将首先判断 `VendState = 'WA'`。对于任何给定的行，仅当州为华盛顿时，这部分的结果才会真；否则为假。接下来，将这部分的结果与字面量`'CA'`执

行 OR 运算。'CA'既不为真，也不为假。数据库系统可能返回错误（对 OR 运算符来说，字符串字面量'CA'是非法的数据类型），可能只返回州为华盛顿的行，也可能先将'CA' OR 'WA'视为布尔表达式进行计算，再将 VendState 同其结果（真或假）进行比较。

在 SELECT 语句中指定条件时，请务必确保它是完整而正确的，否则指定的条件将不管用。

> ❖ **说明**　我通过这个示例演示了使用 OR 运算符时将遇到的一个常见陷阱。然而，如果你认为可使用成员资格条件（如 WHERE VendState IN ('WA', 'CA')）来应答这个请求，这种想法是完全正确的。在有些情况下，表示条件的方式并非只有一种。

3. 结合使用 AND 和 OR

可结合使用 AND 和 OR 来应答棘手的请求。例如，同时使用这两个运算符可应答类似于下面的请求：

“我需要知道区号为 425 且电话号码以 555 打头的员工的姓名，还有 2007 年 10 月 1 日到 12 月 31 日聘用的员工的姓名。”

现在，你应该很容易确定这个请求需要使用哪些类型的条件。你可能已经确定了，为应答这个请求，需要 3 个条件：一个相等比较条件（用于查找区号）、一个模式匹配条件（用于查找电话号码）以及一个范围条件（用于查找 2007 年 10 月 1 日到 12 月 31 日聘用的员工）。现在你只需确定该如何合并这些条件了。

你需要使用 AND 运算符来合并比较条件和模式匹配条件，因为它们指定了你要查找的电话号码，且一行必须同时满足这两个条件才会包含在结果集中。然后，你使用 OR 运算符，将这两个条件组合作为一个整体，与范围条件合并。这样，特定行只要满足合并得到的条件或范围条件，就将包含在结果集中。

下面再次列出了这个请求，还有转换过程：

“我需要知道区号为 425 且电话号码以 555 打头的员工的姓名，还有 2007 年 10 月 1 日到 12 月 31 日聘用的员工的姓名。”

转换	Select first name, last name, area code, phone number, and date hired from the staff table where the area code is 425 and the phone number begins with 555 or the date hired falls between October 1, 2017, and December 31, 2017（从员工表中选择区号为 425 且电话号码以 555 打头或聘用日期在 2007 年 10 月 1 日到 12 月 31 日之间的员工的名、姓、区号、电话号码和聘用日期）
整理	Select first name, last name, area code, phone number, ~~and~~ date hired from ~~the~~ staff ~~table~~ where ~~the~~ area code ~~is~~ = '425' and ~~the~~ phone number ~~begins with~~ like '555%' or ~~the~~ date hired ~~falls~~ between ~~October 1, 2017~~, '2017-10-01' and ~~December 31, 2017~~ '2017-12-31'

SQL

```sql
SELECT StfFirstName, StfLastName, StfAreaCode,
  StfPhoneNumber, DateHired
FROM Staff
WHERE (StfAreaCode = '425'
    AND StfPhoneNumber LIKE '555%')
OR DateHired
    BETWEEN '2017-10-01' AND '2017-12-31'
```

这个示例清楚地演示了这样一种情形，即可在查找条件中使用其他查找条件。我在你转换这个请求前说过，需要使用 AND 运算符将比较条件和模式匹配条件合并，再将它们视为一个整体。根据定义，将合并的一组条件作为一个整体时，它将成为一个查找条件，因此应像这个示例中那样将其放在括号内。然而，需要指出的是，在 SQL 标准和大多数数据库系统中，除按从左到右的顺序处理外，AND 的优先级也高于 OR，因此即便将使用 AND 合并的两个条件放在括号内，也没有关系。务必使用括号来清楚地指出你要如何处理比较。有关运算符的优先级，请参阅 6.3.3 节。

再来看一个结合使用 AND 和 OR 的例子：

“我需要知道 1989 年 5 月 16 日聘用的所有教授和副教授的姓名和职称。”

转换	Select first name, last name, title, and date hired from the staff table where the title is ‘professor’ or ‘associate professor’ and the date hired is May 16, 1989（从教员表中选择职称为教授或副教授且聘用日期为 1989 年 5 月 16 日的员工的名、姓、职称和聘用日期）

整理	Select first name, last name, title, ~~and~~ date hired from ~~the~~ staff ~~table~~ where the title ~~is~~ = 'professor' or title = 'associate professor' and ~~the~~ date hired ~~is~~ = ~~May 16, 1989~~ '1989-05-16'
SQL	SELECT StfFirstName, StfLastName, Title, DateHired FROM Staff WHERE (Title = 'Professor' OR Title = 'Associate Professor') AND DateHired = '1989-05-16'

你可能猜到了，使用 OR 运算符合并的两个条件被视为一个查找条件。这个示例再次强调了这样一个事实：可使用 AND 或 OR 运算符来定义查找条件。需要再次指出的是，关键是确保使用括号将查找条件括起，清楚地指出应如何处理比较。

6.3.2　再谈排除行

如果你觉得有点眼熟，不用担心，这个主题确实讨论过，至少从某种程度上说如此。本章前面说过，NOT 运算符是谓词 BETWEEN、IN、LIKE 和 IS NULL 中的一个选项，但如图 6-11 所示，NOT 也可作为查找条件中的第一个关键字，让你能够将行排除在结果集之外，就像在谓词中使用 NOT 一样。可在单个条件（谓词）**前面**使用 NOT 运算符，也可在嵌套的查找条件前面使用它。再次重申，对于同一个条件，有很多表示方式。

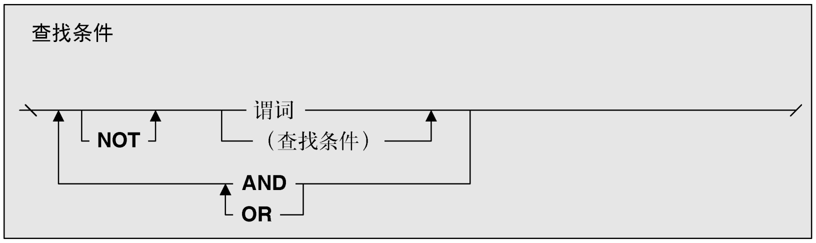


图 6-11　在查找条件中使用 NOT 运算符

假设你要向数据库发出如下请求：

“显示不在 Bolero Lanes、Imperial Lanes 和 Thunderbird Lanes 举行的所有联赛的比赛地点和比赛日期。”

你可能做出了判断，需要使用成员资格条件来应答这个请求。现在你只需确定该如何定义它。一种方法是在谓词中使用 NOT 运算符。

```
WHERE TourneyLocation NOT IN ('Bolero Lanes',
   'Imperial Lanes', 'Thunderbird Lanes')
```

另一种方法是将 NOT 运算符作为查找条件前面的第一个关键字。

```
WHERE NOT TourneyLocation IN ('Bolero Lanes',
   'Imperial Lanes', 'Thunderbird Lanes')
```

这两种方法都将把在 Bolero Lanes、Imperial Lanes 或 Thunderbird Lanes 举行的联赛排除在结果集之外。然而，在查找条件前面使用 NOT 的一个优点是，可将其用于比较条件（前面说过，比较条件的语法不将 NOT 作为可选运算符），这样就可使用比较条件来将行排除在结果集之外了。下面的示例演示了如何使用这种条件：

“列出不居住在贝尔维尤的投球手。”

转换	Select first name, last name, and city from the bowlers table where the city is not ‘Bellevue’（从投球手表中选择城市不为 Bellevue 的投球手的名、姓和城市）
整理	Select first name, last name, ~~and~~ city from ~~the~~ bowlers ~~table~~ where ~~the~~ city ~~is~~ not = 'Bellevue'
SQL	SELECT BowlerFirstName, BowlerLastName, BowlerCity FROM Bowlers WHERE NOT BowlerCity = 'Bellevue'

我知道，这个条件也可表示为 `WHERE BowlerCity <> 'Bellevue'`，但这个示例强调了这样一点，即表示条件的方式有多种。

学习如何在单个条件和整个查找条件中使用 NOT 运算符后，你必须注意一点，那就是如果使用两个 NOT 运算符来定义查找条件，将包含而不是排除行，如下面的示例所示。

“哪些员工不是讲师也不是助教？”

转换	Select first name, last name, and title from the staff table where the title is not ‘teacher’ or ‘teacher’s aide’（从教员表中选择不是讲师也不是助教的教员的名、姓和职称）
整理	Select first name, last name, ~~and~~ title from ~~the~~ staff ~~table~~ where ~~the~~ title ~~is~~ not in ('teacher', ~~or~~ 'teacher''s aide')

SQL	`SELECT StfFirstName, StfLastName, Title`
	`FROM Staff`
	`WHERE NOT Title`
	`NOT IN ('Teacher', 'Teacher''s Aide')`

❖ **说明**　你肯定会问：字符串字面量'Teacher''s Aide'中为何有两个单引号？SQL 标准规定，使用单引号来给字符串或日期时间定界。需要在字符串字面量中嵌入单引号时，必须输入单引号两次来向数据库系统指出这一点。如果不这样做，这个单引号将被视为字符串的结尾定界符，而余下的 s Aide'将导致语法错误。

当然，这里假设你错误地添加了这两个 NOT 运算符中的一个。这条 SELECT 语句依然可以执行，但它将把错误的行放到结果集中。在这里，两个 NOT 运算符相互抵消，就像算术（语言）中的负负得正（双重否定相当于肯定）一样，因此谓词 IN 决定了哪些行将包含在结果集中。其结果是，结果集包含讲师和助教，而非不是讲师和助教的员工。虽然你不会有意这样定义查找条件，但很可能无意间这样做。别忘了，大多数问题是简单错误导致的。

6.3.3　优先级

SQL 标准规定了数据库系统应如何评估查找条件中的各个条件，以及评估的先后顺序。数据库系统如何评估各种条件在本章前面介绍过了，下面介绍数据库系统如何判断评估各个条件的先后顺序。

默认情况下，数据库系统按从左到右的顺序评估各个条件，尤其是在条件比较简单时。在下面的示例中，SELECT 语句首先查找发货日期与下单日期相同的行，再确定其中的哪些行包含顾客编号 1001。然后，将满足这两个条件的行放到结果集中。

SQL	`SELECT CustomerID, OrderDate, ShipDate`
	`FROM Orders`
	`WHERE ShipDate = OrderDate`
	`    AND CustomerID = 1001`

要让这条 SELECT 语句先查找特定的顾客编号，再评估发货日期，只需将这两个条件交换位置即可。有关你想这样做的原因，将在本节后面讨论。

查找条件包含各种不同的条件时，数据库系统将根据每个条件使用的运算符按特定的顺序评估它们。SQL 标准定义的运算符优先级如下。

优　先　级	运　算　符
1	正号（+）、负号（–）
2	乘法（*）、除法（/）
3	加法（+）、减法（–）
4	=、<>、<、>、<=、>=、BETWEEN、IN、LIKE、IS NULL
5	NOT
6	AND
7	OR

下面的 SELECT 语句包含的查找条件导致数据库系统根据优先级评估各个条件。在这个示例中，数据库系统将先执行加法运算，再进行比较，然后判断是否至少满足其中一个条件，最后将至少满足一个条件的行放到结果集中。

```
SQL          SELECT CustomerID, OrderDate, ShipDate
             FROM Orders
             WHERE CustomerID = 1001
               OR ShipDate = OrderDate + 4
```

1. 提高条件的优先级

明白优先级后，就可极大地提高查找条件的准确度，这可帮助你根据请求编写正确的条件。然而，你必须小心避免定义模糊的条件，因为它们生成的结果可能出乎意料。

下面通过一个示例来看看这种潜在的问题：

```
SQL          SELECT CustFirstName, CustLastName, CustState,
                 CustZipCode
             FROM Orders
             WHERE CustLastName = 'Patterson'
                 AND CustState = 'CA'
                 OR CustZipCode LIKE '%9'
```

在这里，难以判断查找条件的真正意图，因为对其有两种解读。

- 查找位于加州且名字为 Patterson 的人，还有邮政编码以 9 结尾的人。
- 查找名字为 Patterson 的人，还有位于加州或邮政编码以 9 结尾的人。

如果你还记得前面的优先级表，就知道第一种解读是正确的，因为数据库系统应先评估 AND 再评估 OR。但你总是能够记住评估顺序吗？为避免这种二义性并让查找条件更清晰，可使用括号来合并某些条件并提高其优先级。例如，基于对查找条件的第一种解读，可这样定义 WHERE 子句：

```
WHERE (CustLastName = 'Patterson' AND CustState = 'CA') OR
CustZipCode LIKE '%9'
```

括号确保数据库系统先分析和评估两个比较条件，再分析和评估模式匹配条件。

你也可以采取第二种解读，像下面这样定义 WHERE 子句：

```
WHERE CustLastName = 'Patterson' AND (CustState = 'CA' OR
CustZipCode LIKE '%9')
```

在这种情况下，数据库系统将先分析和评估第二个比较条件和模式匹配条件，再分析和评估第一个比较条件。

你早已熟悉将条件放在括号内的概念，因为本章前面讨论合并条件时就介绍了如何这样做。这里要强调的是，括号的位置会严重影响查找条件的结果。

你可根据需要将任意数量的条件括起，还可在条件中嵌套条件。与表达式一样，查找条件也是按从左到右、从内到外的顺序处理的，但有多个条件位于相同的层级时，数据库系统将先处理 AND 再处理 OR。下面说明了数据库系统如何处理包含括号的查找条件。

- 先处理括号内的查找条件，再处理括号外的查找条件。
- 对于位于括号内的多个查找条件，按从左到右的顺序处理。
- 对于多个使用括号嵌套的处理条件，按从内到外的顺序处理。

对给定的位于括号内的条件进行分析时，数据库系统根据正常的优先级顺序计算其中的各个表达式。只要仔细地转换请求，并在查找条件中有效地使用括号，就能得到更好的结果。

2. 少比多好

本节开头说过，数据库系统按从左到右的顺序评估条件，并在遇到复杂的条件时根据优先级进行评估。我还说过，你在查找条件中使用括号的方式将直接影响查找条件的结果。下面介绍一种简单而通用的技巧，使用它可提高查找条件的处理速度。这种技巧就是要求不要太多：只选择能够满足请求的列，并让查找条件尽可能具体，从而让数据库系

统处理的行尽可能少。需要使用多个条件时，确保先处理能够将大部分列排除在结果集之外的条件，这样数据库系统就可能会更快地找到答案（此时你对优先级的认识将大有裨益）。

下面通过本节前面使用的一个示例来演示这种技巧。

SQL

```
SELECT CustomerID, OrderDate, ShipDate
FROM Orders
WHERE ShipDate = OrderDate
    AND CustomerID = 1001
```

在这里，特定行必须同时满足两个条件才会包含在结果集中。这种谓词排列顺序让数据库系统先查找发货日期与下单日期相同的订单；根据表中包含的行数，数据库系统可能需要花较长的时间来评估这个条件。然后，数据库系统在满足第一个条件的行中找出顾客 ID 为 1001 的行。

下面这种定义条件的方式可能更佳：

SQL

```
SELECT CustomerID, OrderDate, ShipDate
FROM Orders
WHERE CustomerID = 1001
    AND ShipDate = OrderDate
```

这样数据库系统很可能先查找顾客 ID，而根据这个条件生成的行数很可能更少，这意味着数据库系统为查找发货日期与订单日期相同的行所需的时间更少。

你应将这种技巧视为惯常做法，并在定义查找条件时使用它，这对确保 SELECT 语句快速而高效地执行大有裨益。请务必研究你使用的数据库系统的文档，学习可用来进一步优化 SELECT 语句的其他技巧。

❖ **说明** 几乎所有商业数据库系统都包含查询优化器，它查看整个请求，力图找出返回答案的最快方式。在决定优化器选择如何做方面，数据库管理员在表列上定义的索引影响最大，但通过将最严格的查找条件放在最前面，可进一步影响数据库系统的优化器，因此养成这样的习惯没有害处。

明白如何合并查找条件后，下面顺便说说一个更复杂的问题。如果你要通过比较两个范围来查找满足条件的行，该怎么办呢？请接着往下读！

6.3.4 检查两个范围是否重叠

需要检查某列的值是否在特定范围内时，BETWEEN 确实很好用。你还知道，可检查指定的值是否在两个起始/结束（高/低）表列定义的范围内。但如果你要确定两个范围是否重叠，该怎么办呢？例如，你可能想获悉所有这样的演出合约，即演出时间与 2017 年 11 月 12 日到 18 日这一周重叠（每个演出都有开始时间和结束时间）。你可能想使用 BETWEEN 来解决这个问题，如下所示。

“列出演出时间与 2017 年 11 月 12 日到 18 日这一周重叠的演出合约。”

转换	Select engagement number, start date, and end date from the engagements table where start date is between November 12, 2017, and November 18, 2017 and end date is between November 12, 2017, and November 18, 2017（从演出合约表中选择开始时间和结束时间都在 2017 年 12 月 12 日和 12 月 18 日之间的演出合约的编号、开始时间和结束时间）
整理	Select engagement number, start date, ~~and~~ end date from ~~the~~ engagements ~~table~~ where start date ~~is~~ between ~~November 12, 2017~~ '2017-11-12' and ~~November 18, 2017~~ '2017-11-18' and end date is between ~~November 12, 2017~~ '2017-11-17' and ~~November 18, 2017~~ '2017-11-18'

SQL

```
SELECT EngagementNumber, StartDate, EndDate
FROM Engagements
WHERE StartDate
    BETWEEN '2017-11-12' AND '2017-11-18'
AND EndDate BETWEEN '2017-11-12' AND '2017-11-18'
```

很接近，但不对。你要查找的是有部分演出时间落在 11 月的两个日期之间的演出合约。简单的 BETWEEN 子句组合不管用，要明白其中的原因，请看图 6-12。

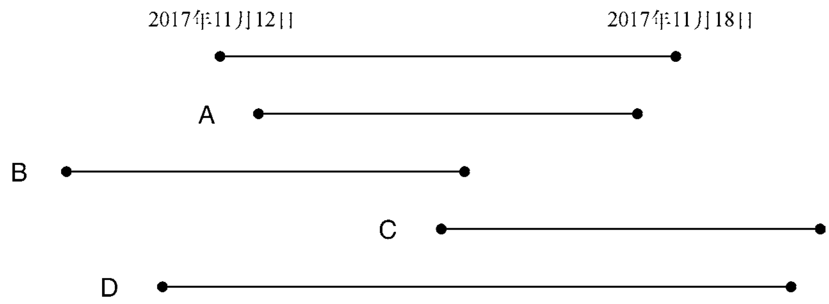


图 6-12　有部分演出时间落在指定时段内的演出合约

从该图可知，有 4 种可能的演出时间跨度，它们完全或部分落在指定的那周内。

- 有些演出完全落在指定的时间跨度内，如线条 A 所示。
- 有些演出的开始时间在指定时间跨度之前，但结束时间在该时间跨度内，如线条 B 所示。
- 有些演出的开始时间落在该时间跨度内，但结束时间在该时间跨度之后，如线条 C 所示。
- 最后，有些合约的开始时间在该时间跨度之前，而结束时间在该时间跨度之后，如线条 D 所示。

如果你仔细研究前面的 SQL，将发现它查找的只是类似于线条 A 的演出合约。类似于 B 的演出合约因为开始日期不在 2017 年 11 月 12 日到 18 日内而被排除在外，虽然它有部分演出时间在这个时间跨度内；类似于 C 的演出因为结束时间不在指定的两个日期之间而被排除在外；类似于 D 的演出合约因为开始日期和结束日期都不在指定范围内而被排除在外，虽然其演出时间有一部分在该范围内。

那么如何解决这个问题呢？可针对每种可能的情形编写一个查找条件，如下所示：

```
WHERE (StartDate BETWEEN '2017-11-12' AND '2017-11-18'
AND EndDate BETWEEN '2017-11-12' AND '2017-11-18')
OR (StartDate <= '2017-11-12')
AND EndDate BETWEEN '2017-11-12' AND '2017-11-18')
OR (StartDate BETWEEN '2017-11-12' AND '2017-11-18'
AND EndDate >= '2017-11-18')
OR (StartDate <= '2017-11-12'
AND EndDate >= '2017-11-18')
```

是不是不太好看？再来看一眼图 6-12，所有开始日期都有一个什么共同之处呢？它们都小于等于指定日期跨度的终点！同样，它们的结束日期都大于等于指定日期跨度的起点。因此，最简单的答案如下：

```
SQL         SELECT EngagementNumber, StartDate, EndDate
            FROM Engagements
            WHERE StartDate <= '2017-11-18'
            AND EndDate >= '2017-11-12'
```

是不是简单得多？请记住这个解决方案，完成本章末尾的练习时要用到。现在言归正传，再来谈谈 Null。

6.4　再谈 Null：一个注意事项

现在是说说 Null 的绝佳时机。第 5 章说过，Null 表示值缺失，而包含 Null 的表达式的结果为 Null。查找条件的情况也如此——评估 Null 的谓词的结果**不可能为真**，但其结果也不可能为假——这可能令人迷惑！SQL 标准规定，评估 Null 的谓词的结果都是未知的。别忘了，仅当谓词的结果为真时，才会选择相应的行，因此结果为假或未知都将导致相应的行被拒绝。

为帮助澄清这一点，再来看看图 6-9 和图 6-10 所示的真值表。图 6-13 和图 6-14 再次列出了它们，但包含涉及 Null 时结果未知的情况。

第二个表达式（列）；第一个表达式（行）

AND	真	假	未知
真	真（行被选择）	假（行被拒绝）	未知（行被拒绝）
假	假（行被拒绝）	假（行被拒绝）	假（行被拒绝）
未知	未知（行被拒绝）	假（行被拒绝）	未知（行被拒绝）

图 6-13 使用 AND 运算符合并两个谓词表达式［其中一个为 Null（未知）］的结果

第二个表达式（列）；第一个表达式（行）

OR	真	假	未知
真	真（行被选择）	真（行被选择）	真（行被选择）
假	真（行被选择）	假（行被拒绝）	未知（行被拒绝）
未知	真（行被选择）	未知（行被拒绝）	未知（行被拒绝）

图 6-14 使用 OR 运算符合并两个谓词表达式［其中一个为 Null（未知）］的结果

如你所见，评估 Null 列的谓词的结果为未知，这破坏了大局！例如，假设有一个简单的比较谓词——A = B，如果对于给定的行，A 或 B 为 Null，比较的结果将是未知的。由于结果不为真，这行将被拒绝。如果 A = B 不为真，你可能认为 NOT（A = B）为真。错！结果也是未知。图 6-15 可帮助你明白为何如此。

（表达式）	NOT（表达式）
真	**假**
假	**真**
未知	**未知**

图 6-15　将 NOT 应用于真/假/未知值的结果

假设你要向数据库发出如下请求：

“显示姓 Hernandez 的 King 县居民的姓名和电话号码。”

转换	Select first name, last name, and phone number from the customers table where the county name is ‘King’ and the last name is ‘Hernandez’（从顾客表中选择县名为 King、姓为 Hernandez 的顾客的名、姓和电话号码）
整理	Select first name, last name, ~~and~~ phone number from ~~the~~ customers ~~table~~ where ~~the~~ county ~~name is~~ = 'King' and ~~the~~ last name ~~is~~ = 'Hernandez'

SQL	`SELECT CustFirstName, CustLastName,`
	`   CustPhoneNumber`
	`FROM Customers`
	`WHERE CustCounty = 'King'`
	`   AND CustLastName = 'Hernandez'`

你知道，一行要出现在结果集中，必须同时满足这两个条件。对于县名或姓为 Null 的行，数据库系统根本就不考虑。请看下面的请求：

“列出所有为研究生辅导员（graduate counselor）或 2007 年 9 月 1 日聘用的教员的姓名。”

转换	Select last name and first name from the staff table where the title is ‘graduate counselor’ or date hired is September 1, 2007（从教员表中选择职位为 graduate counselor 或聘用日期为 2007 年 9 月 1 日的教员的姓和名）
整理	Select last name ~~and~~ first name from ~~the~~ staff ~~table~~ where ~~the~~ title ~~is~~ = 'graduate counselor' or date hired ~~is~~ = ~~September 1,2007~~ '2007-09-01'

SQL	`SELECT StfLastName, StfFirstName`
	`FROM Staff`
	`WHERE Title = 'Graduate Counselor'`
	`   OR DateHired = '2007-09-01'`

你可能认为，对于使用 OR 合并的条件，Null 的影响与使用 AND 合并条件时相同，但情况并非如此：只要满足其中的条件之一，一行就有机会出现在结果集中。请再看一眼图 6-14。表 6-2 说明了当你使用 OR 来合并谓词时，数据库系统将如何根据职位和聘用日期决定是否将给定行放到结果集中。

表 6-2　Determining the Result Set with OR

职　　位	聘用日期	结　　果
Graduate Counselor	2007-09-01	这行将出现在结果集中，因为它同时满足两个条件
Graduate Counselor	2007-11-15	这行将出现在结果集中，因为它满足第一个条件
Registrar	2007-09-01	这行将出现在结果集中，因为它满足第二个条件
Graduate Counselor	Null	这行将出现在结果集中，因为它满足第一个条件
Null	2007-09-01	这行将出现在结果集中，因为它满足第二个条件
Null	Null	这行将被排除在结果集之外，因为它不满足任何一个条件

怀疑结果集显示的信息不正确时，请使用 Null 值来测试所有用作条件的列。这将给你提供妥善处理 Null 值的机会，然后再次执行原来的 SELECT 语句。例如，如果你认为结果集中可能缺失了一些研究生辅导员，可执行下面的 SELECT 语句来确定是否确实如此：

```
SQL         SELECT StfLastName, StfFirstName, Title
            FROM Staff
            WHERE Title IS NULL
```

如果有一些行的 Title 列的值为 Null，在这条 SELECT 语句生成的结果集中，将包含所有没有指定职位的教员的姓名。现在你可妥善地处理这些数据，再执行原来的 SELECT 语句。

对 Null 的讨论不会就此止步，第 12 章介绍汇总数据的 SELECT 语句时，将再次讨论 Null。

6.5 以不同的方式表示条件

学习本章内容的一个额外好处是，现在对于给定的条件，你能够以不同的方式表示它。下面通过示例来说明这一点，请看如下请求：

“列出 2007 年 10 月聘用的所有教员的姓名。”

为应答这个请求，需要查找落在 2007 年 10 月 1 日到 31 日内的聘用日期。基于你学到的知识，可以以如下两种方式定义这个条件：

```
DateHired BETWEEN '2007-10-01' AND '2007-10-31'
DateHired >= '2007-10-01' AND DateHired <= '2007-10-31'
```

这两种方式在结果集中包含的行相同，具体使用哪种方式完全取决于个人喜好。有些人认为第一种表示方式更容易理解，但有些人更喜欢第二种表示方式。

下面再来看几个等价条件示例。

“列出位于加州、俄勒冈州或华盛顿州的供应商。”

```
VendState IN ('CA', 'OR', 'WA')
VendState = 'CA' OR VendState = 'OR' OR VendState = 'WA'
```

“列出姓以 H 打头的顾客。”

```
CustLastName >= 'H' AND CustLastName <= 'HZ'
CustLastName BETWEEN 'H' AND 'HZ'
CustLastName LIKE 'H%'
```

“列出所有不居住在西雅图和雷德蒙德的学生。”

```
StudCity <> 'Seattle' AND StudCity <> 'Redmond'
StudCity NOT IN ('Seattle', 'Redmond')
NOT (StudCity = 'Seattle' OR StudCity = 'Redmond')
```

定义条件的方式没有对错之分，但如果你无视语法，定义的条件就可能是错误的（你知道，这可能导致条件不满足）。然而，对于有些类型的条件，有些数据库系统会对其进行优化以提高处理速度，这让这些类型的条件优于其他等价条件。请参阅你使用的数据库系统的文档，了解它偏爱哪种条件定义方法。

6.6 语句举例

至此，你学习了编写可靠查找条件所需的所有技巧，下面来看一些查找条件示例，它们使用了示例数据库中的表。这些示例演示了如何使用查找条件来筛选数据。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站（www.informit.com/title/9780134858333）提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库，查询名以 CH06 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

❖ **说明**　在下面的示例中，我将转换和整理合二为一了，以便你继续练习合并后的流程。

- **Sales Orders 数据库**

"列出编号为 1001 的顾客的所有订单。"

转换/整理	Select ~~the~~ order number ~~and~~ customer ID from ~~the~~ orders ~~table~~ where ~~the~~ customer ID ~~is equal to~~ = 1001（从订单表中选择 ID 为 1001 的顾客的订单号和顾客 ID）

SQL	`SELECT OrderNumber, CustomerID`
	`FROM Orders`
	`WHERE CustomerID = 1001`

CH06_Orders_for_Customer_1001（44 行）

OrderNumber	CustomerID
2	1001
7	1001
16	1001
52	1001
55	1001
107	1001
137	1001
138	1001
151	1001
154	1001
<< 其他行 >>	

"按字母顺序列出名称以 Dog 打头的商品。"

转换/整理	Select ~~the~~ product name from ~~the~~ products ~~table~~ where ~~the~~ product name like 'Dog%' ~~and~~ order by product name（从商品表中选择名称类似于 Dog%的商品的名称，并按商品名排序）

SQL	`SELECT ProductName`
	`FROM Products`
	`WHERE ProductName LIKE 'Dog%'`
	`ORDER BY ProductName`

CH06_Products_That_Begin_With_DOG（4 行）

ProductName
Dog Ear Aero-Flow Floor Pump
Dog Ear Cyclecomputer
Dog Ear Helmet Mount Mirrors
Dog Ear Monster Grip Gloves

❖ **说明**　需要提醒你的是，请将 ORDER BY 子句放在 SELECT 语句末尾。如果必要，请复习 4.6 节。另外别忘了，在 Microsoft SQL Server 中，如果你执行保存在视图中的 SELECT 语句，其中的 ORDER BY 子句将被忽略。要看到上面的排列顺序，你必须在设计器中打开相应的视图，再在那里执行它。

- **Entertainment Agency 数据库**

“按字母顺序列出位于贝尔维尤、雷德蒙德或伍丁维尔的演唱组合。”

转换/整理	Select stage name, phone number, ~~and~~ city from ~~the~~ entertainers ~~table~~ where ~~the~~ city ~~is~~ in ('Bellevue', 'Redmond', ~~or~~ 'Woodinville') ~~and~~ order by stage name（从演唱组合表中选择城市为 Bellevue、Redmond 或 Woodinville 的演唱组合的艺名、电话号码和城市，并按艺名排序）
SQL	SELECT EntStageName, EntPhoneNumber, EntCity FROM Entertainers WHERE EntCity IN ('Bellevue', 'Redmond', 'Woodinville') ORDER BY EntStageName

CH06_Eastside_Entertainers（7 行）

EntStageName	EntPhoneNumber	EntCity
Carol Peacock Trio	555-2691	Redmond
Jazz Persuasion	555-2541	Bellevue
Jim Glynn	555-2531	Bellevue
JV & the Deep Six	555-2511	Redmond
Katherine Ehrlich	555-0399	Woodinville
Modern Dance	555-2631	Woodinville
Susan McLain	555-2301	Bellevue
	<< 其他行 >>	

“列出所有持续 4 天的演出合约。”

转换/整理	Select engagement number, start date, ~~and~~ end date from ~~the~~ engagements ~~table~~ where ~~the~~ CAST(end date ~~minus~~ - start date AS INTEGER) ~~is equal to~~ = 3（从演出合约表中选择结束日期与开始日期相差 3 天的演出合约的编号、开始日期和结束日期）
SQL	SELECT EngagementNumber, StartDate, EndDate FROM Engagements WHERE CAST(EndDate - StartDate AS INTEGER) = 3

CH06_FourDay Engagements（15 行）

EngagementNumber	StartDate	EndDate
5	2017-09-12	2017-09-15
13	2017-09-18	2017-09-22
17	2017-09-30	2017-10-03
21	2017-10-01	2017-10-04
56	2017-11-26	2017-11-29
58	2017-12-02	2017-12-05
59	2017-12-02	2017-12-05
63	2017-12-19	2017-12-22
70	2017-12-24	2017-12-27
95	2018-01-16	2018-01-19
	<< 其他行 >>	

❖ **说明**　演出从开始日期持续到结束日期，将 StartDate 减去 EndDate 得到的天数比演出持续的天数少 1，因此我将相减的结果同 3 而不是 4 比较。

- **School Scheduling 数据库**

“按字母顺序列出年薪为 40 000~50 000 美元的所有教员及其薪水。”

转换/整理　Select first name, last name, ~~and~~ salary from ~~the~~ staff ~~table~~ where ~~the~~ salary ~~is~~ between 40000 and 50000, ~~then~~ order by last name, ~~and~~ first name（从教员表中选择年薪为 40 000~50 000 美元的教员的名、姓和薪水，并按姓和名排序）

SQL
```
SELECT StfFirstName, StfLastName, Salary
FROM Staff
WHERE Salary BETWEEN 40000 AND 50000
ORDER BY StfLastname, StfFirstName
```

CH06_Staff_Salaries_40K_T0_50K（14 行）

StfFirstName	StfLastName	Salary
Robert	Brown	$49,000.00
Kirk	DeGrasse	$45,000.00
Katherine	Ehrlich	$45,000.00
Jim	Glynn	$45,000.00
Liz	Keyser	$48,000.00
Ann	Patterson	$45,000.00
Maria	Patterson	$48,000.00
Mariya	Sergienko	$45,000.00
Tim	Smith	$40,000.00
Caleb	Viescas	$45,000.00
<< 其他行 >>		

“列出姓为 Kennedy 或居住在西雅图的学生。”

转换/整理　Select first name, last name, ~~and~~ city from ~~the~~ students ~~table~~ where ~~the~~ last name ~~is~~ = 'Kennedy' or ~~the~~ city ~~is~~ = 'Seattle'（从学生表中选择姓为 Kennedy 或城市为 Seattle 的学生的名、姓和城市）

SQL
```
SELECT StudFirstName, StudLastName, StudCity
FROM Students
WHERE StudLastName = 'Kennedy'
  OR StudCity = 'Seattle
```

CH06_Seattle_Students_And_Students_Named_Kennedy（4 行）

StudFirstName	StudLastName	StudCity
Doris	Hartwig	Seattle
John	Kennedy	Portland
Kendra	Bonnicksen	Seattle
Richard	Lum	Seattle

- **Bowling League 数据库**

“列出在前 10 场比赛第 3 局获胜过的球队的 ID。”

转换/整理　Select the team ID, match ID, ~~and~~ game number from ~~the~~ match_games ~~table~~ where ~~the~~ game number ~~is~~ = 3 and ~~the~~ match ID ~~is~~ between 1 and 10（从场次局次表中选择局次为 3 且场次为 1~10 的球队 ID、场次和局次）

SQL
```
SELECT WinningTeamID, MatchID, GameNumber
FROM Match_Games
WHERE GameNumber = 3
AND MatchID BETWEEN 1 AND 10
```

CH06_Game3_Top_Ten_Matches（10 行）

WinningTeamID	MatchID	GameNumber
1	1	3
3	2	3
5	3	3
7	4	3
3	5	3
4	6	3
5	7	3
8	8	3
2	9	3
1	10	3

“列出 3、4 和 5 号球队中姓以字母 H 打头的投球手。”

转换/整理	Select first name, last name, ~~and~~ team ID from ~~the~~ bowlers ~~table~~ where ~~the~~ team ID ~~is either~~ in (3, 4, ~~or~~ 5) and ~~the~~ last name ~~begins with the letter~~ like 'H%'（从投球手表中选择球队 ID 为 3、4 或 5 且姓类似于 H%的投球手的名、姓和球队 ID）

SQL	`SELECT BowlerFirstName, BowlerLastName,`
	`TeamID`
	`FROM Bowlers`
	`WHERE (TeamID IN (3,4,5))`
	`   AND (BowlerLastName LIKE 'H%')`

CH06_H_Bowlers_Teams_3_Through_5（4 行）

BowlerFirstName	BowlerLastName	TeamID
Elizabeth	Hallmark	4
Gary	Hallmark	4
Kendra	Hernandez	5
Michael	Hernandez	5

- **Recipes 数据库**

“列出没有说明的菜品。”

转换/整理	Select ~~the~~ recipe title from ~~the~~ recipes ~~table~~ where notes is ~~empty~~ Null（从菜品表中选择说明为空的菜品的名称）

SQL	`SELECT RecipeTitle`
	`FROM Recipes`
	`WHERE Notes IS NULL`

CH06_Recipes_With_No_Notes（6 行）

RecipeTitle
Irish Stew
Salsa Buena
Fettuccini Alfredo
Mike’s Summer Salad
Roast Beef
Yorkshire Pudding

“列出属于肉食（2 类食材）但不是鸡肉的食材。”

转换/整理	Select ingredient name from ~~the~~ ingredients ~~table~~ where ingredient class ID ~~is equal to~~ = 2 and ingredient name ~~does~~ not ~~contain~~ like ‘%chicken%’（从食材表中选择类 ID 为 2，但名称中不包含 chicken 的食材的名称）

SQL	`SELECT IngredientName`
	`FROM Ingredients`
	`WHERE (IngredientClassID = 2)`
	`   AND (IngredientName NOT LIKE '%chicken%')`

CH06_Meats_That_Are_Not_Chicken（5 行）

IngredientName
Beef
Bacon
T-bone Steak
New York Steak
Ground Pork

❖ **说明**　PostgreSQL 是区分大小写的，因此上述查询不管用，因为所有鸡肉类食材的名称中包含的都是 Chicken，而不是 chicken。在 PostgreSQL 中，务必使用条件 `IngredientName NOT LIKE '%Chicken%'`。当然，可能有名为“Ground chicken”的食材，因此将字母 c 改成大写后，将不与这种食材匹配。为保险起见，必须再添加一个谓词，以同时检查大写和小写。在 SQL Server 和 MySQL 中，如果你安装它们时启用了区分大小写选项，也必须这样做（这两个数据库系统默认都不区分大小写）。

6.7　小结

本章介绍了如何在 WHERE 子句中使用查找条件来过滤包含在结果集中的信息。查找条件使用谓词组合来筛选要放到结果集中的数据，而谓词是可对值表达式执行的特定检查。随后介绍了 5 种基本谓词。

接下来深入探讨了可在 WHERE 子句的查找条件中定义的 5 种基本谓词。你学习了如何对值进行比较，以及如何检查一个值是否在指定范围内；如何检查一个值是否与一组值中的某个匹配，以及它是否与指定的模式字符串匹配。另外，你还了解到，可使用 NOT 运算符将行排除在结果集之外。

然后讨论了如何使用 AND 和 OR 运算符来合并多个条件。你了解到，一行必须满足使用 AND 合并的所有条件，才会出现在结果集中；而如果条件是使用 OR 合并的，一行只需满足其中一个条件就会出现在结果集中。你还学习了如何结合使用 AND 和 OR 来应答复杂的请求。接下来，我们重温了如何使用 NOT 来将行排除在结果集之外，而你了解到可在查找条件的两个不同层级中使用 NOT。

接下来讨论了优先级，你学习了数据库系统是如何对条件进行分析和评估的。现在你知道，数据库系统根据每个条件使用的运算符按特定的顺序评估它们。你还学习了如何使用括号来修改数据库系统评估条件的顺序，以及如何使用括号来避免定义模糊的条件。

我们顺便简单地介绍了如何判断两个范围彼此重叠；解决之道非常简单，不需要使用 BETWEEN。

接下来再次讨论了 Null。你了解到，Null 对条件的影响几乎与其对表达式的影响相同。你还知道，如果结果集显示的信息不正确，应检查 Null 值。

最后讨论了这样一点：对于给定的条件，可以以各种不同的方式表示它。例如，要查找姓以字母 H 打头的人，可使用 3 种不同的条件。

本书的下一部分将介绍集以及可对其执行的运算类型。学习集后，就为学习如何使用多个表定义 SELECT 语句打下了坚实的基础。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

6.8 练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 列出所有位于巴拉德、贝尔维尤或雷德蒙德的供应商的名称。

  解决方案见 CH06_Ballard_Bellevue_Redmond_Vendors（3 行）。

  (2) 按字母顺序列出零售价为 125.00 美元或更高的商品。

  提示：要按字母顺序排列，可使用前一章讨论的一个子句。

  解决方案见 CH06_Products_Priced_Over_125（13 行）。

  (3) 与我们合作的哪些供应商没有网站？

  解决方案见 CH06_Vendors_With_No_Website（4 行）。

- **Entertainment Agency 数据库**

  (1) 给我一个清单，列出所有在 2017 年 10 月演出的演出合约。

  提示：为解决这个问题，需要检查表中的一个范围是否至少有一部分落在另一个范围，即 10 月的第 1 天到最后 1 天内。

  解决方案见 CH06_October_2017_Engagements（24 行）。

  (2) 列出演出在 2017 年 10 月进行且开始时间为正午到下午 5 点的演出合约。

  解决方案见 CH06_October_Dates_Between_Noon_and_Five（17 行）。

  (3) 列出在同一天开始和结束的所有演出合约。

  解决方案见 CH06_Single_Day_Engagements（5 行）。

- **School Scheduling 数据库**

  (1) 告诉我哪些员工将邮政信箱作为地址。

  解决方案见 CH06_Staff_Using_POBoxes（3 行）。

  (2) 你能告诉我哪些学生不居住在太平洋西北地区吗？

  解决方案见 CH06_Students_Residing_Outside_PNW（5 行）。

  (3) 列出所有科目代码以 MUS 打头的科目。

  解决方案见 CH06_Subjects_With_MUS_In_SubjectCode（4 行）。

  (4) 生成一个清单，列出所有专职副教授的 ID。

  解决方案见 CH06_Full_Time_Associate_Professors（4 行）。

- **Bowling League 数据库**

  (1) 给我一个清单，列出在 2017 年 9 月举行的联赛。

  解决方案见 CH06_September_2017_Tournament_Schedule（4 行）。

  (2) 告诉我在保龄球馆 Bolero Lanes、Red Rooster Lanes 和 Thunderbird Lanes 举行的联赛的日程安排。

  解决方案见 CH06_Eastside_Tournaments（9 行）。

  (3) 列出居住在东区（即贝尔维尤、博塞尔、杜瓦尔、雷德蒙德和伍丁维尔）且属于 5、6、7 或 8 号球队的投球手。

  提示：使用 IN 来查找城市，使用 BETWEEN 来查找球队编号。

  解决方案见 CH06_Eastside_Bowlers_On_ Teams_5_Through_8（9 行）。

- **Recipes 数据库**

  (1) 列出属于主菜（即 1 类菜品）且有说明的菜品。

  解决方案见 CH06_Main_Courses_With_Notes（4 行）。

  (2) 显示前 5 种菜品。

  提示：使用 BETWEEN 检查相应表的主键。

  解决方案见 CH06_First_5_Recipes（5 行）。

第三部分

使用多个表

第 7 章 集合思维

“小声喝彩，盛宴款待。”

——威廉·莎士比亚《错误的喜剧》第一幕第三场

本章涵盖如下主题：

- 何谓集合
- 集合运算
- 交集
- 差集
- 并集
- SQL 集合运算
- 小结

现在，你知道如何通过请求特定列或使用包含列的表达式来生成一组信息（SELECT）、如何对行进行排序（ORDER BY）以及如何限制返回的行（WHERE）。本书前面执行的都是基本操作，只涉及一个表，但如果要获取多个表中的信息，该怎么办呢？如果要对来自同一个表或不同表的信息进行比较，又该怎么办呢？

将一堆土豆或胡萝卜去皮、切片并炒出一道菜很容易。从现在开始，我列举的大多数问题需要从多个表中获取数据才能解决。我不仅会演示如何做出一道好菜，还会演示如何成为主厨。

阅读本章前你必须知道，本章将介绍各种概念，你必须理解它们才能将多组信息关联起来。本章还将概述 SQL 标准为支持这些概念而定义的具体语法，但请注意，当前很多 SQL 商业实现都不支持这些“纯粹”的语法。本书后面将介绍如何使用大多数主流数据库系统支持的 SQL 语法来实现这些概念，而本章要探讨的是法的精神，而不是法律条文。

7.1 何谓集合

如果你是 20 世纪 60 年代中期的中小学生，可能在数学课上学过集合论。[还记得新数学（New Math）吗？你的年龄可能没这么大！] 如果你上过集合代数课，很可能心存疑惑——这有什么用呢？

你学习关系型数据库和古怪的 SQL 语言，旨在创建应用程序、解决问题或消除心中的疑惑。在代数课上，你认真听了吗？如果认真听了，你在使用 SQL 解决问题（尤其是复杂的问题）时，将容易得多。

实际上，从开始阅读本书起，你就一直在与集合打交道。在第 1 章，你学习了关系型数据库的基本结构——表，而表由包含一列或多列的行组成。数据库中的每个表都是有关某种客体的信息集。在第 2 章，你学习了如何核实数据库的结构是否合理。每个表都应包含且只包含有关特定客体的信息集。

在第 4 章，你学习了如何编写基本的 SELECT 语句以生成包含单个表中特定列的结果集，以及如何对结果集进行排序。在第 5 章，你学习了如何编写表达式，通过操作表中一个或多个列来生成新的信息集。在第 6 章，你学习了如何在查询中添加筛选器（WHERE 子句），以限制从表中检索的信息集。

如你所见，集合可以很小，小到只包含表中一行中某列的值。实际上，使用 SQL 可编写不返回任何行（即返回一个空集）的请求。在有些情况下，确定某样东西不存在很有用。集合也可包含多个表的多行中的多列（包括使用表达式生成的列）。结果集中的每行都是一个集**成员**，而列值是集成员的特定**属性**——描述集成员的数据项。在接下来的几章中，我将介绍如何从多个数据集中获取信息以及如何将这些数据集关联起来，这可帮助你回答较为复杂的问题。但在此之前，你需要更深入地了解集合以及合并集合的逻辑方式。

7.2 集合运算

第 1 章说过，E. F. Codd 博士发明了关系模型，而大多数现代数据库以及 SQL 是基于这个模型的。这个新模型的基石是两个数学分支——集合论和一阶谓词逻辑。

如果不想局限于在单个表中寻找答案，需要学习如何使用结果集来解决复杂的问题。这些复杂的问题通常要求使用常见的集合运算将多个表中的数据关联起来。在有些情况下，为了得到答案，需要从同一个表获取两个不同的结果集，再将它们合并。

下面是 3 种最常见的集合运算。

- **交集**：用于找出多个不同的集合中共有的元素，如列出所有的学生及其注册的课程、显示使用了羊肉和大米的菜品、列出订购了自行车和头盔的顾客。
- **差集**：用于找出出现在一个集合中但未出现在另一个集合中的元素，如列出使用了羊肉但没使用大米的菜品、列出订购了自行车但未订购头盔的顾客。
- **并集**：用于合并多个类似的集合，如列出使用了羊肉或大米的菜品、列出订购了自行车或头盔的顾客、列出职员和学生的姓名和地址。

接下来的三小节将分别介绍这 3 种基本的集合运算（你在高中代数课上应该学过）。7.3 节将概述这些运算在 SQL 标准中是如何实现的。

7.2.1 交集

两个集合的**交集**包含这两个集合都有的元素。下面先从集合论的角度介绍交集，再介绍如何使用交集来解决业务问题。

1. 集合论中的交集

交集是科学家和工程师常用的功能强大的数学工具。科学家可能想找出两组化学或物理样本数据中都有的点。例如，从事药剂研究的化学家可能发现两种药剂看起来有相同的效果，通过找出它们的共同之处（交集）将有助于确定是什么成分让这两种药剂有效。工程师可能想找出两种合金都有的成分，它们一个硬而脆，另一个软而富有弹性。

下面来看看如何计算两个数字集合的交集。在这个示例中，每个数字都是集合的一个成员。第一个数字集合如下：

1, 5, 8, 9, 32, 55, 78

第二个数字集合如下：

3, 7, 8, 22, 55, 71, 99

这两个数字集合的交集包含两个集合中都有的数字：

8, 55

集合的各个成员并非只能包含单个值。实际上，使用 SQL 解决问题时，涉及的可能是行集。

集合论指出，一个集合的成员并非只包含一个数字或值时，这个集合的成员便有多个属性或多项数据，它们描述了成员的特征。例如，在由所有菜品组成的集合中，你喜欢的菜品是一个复合成员，而制作这道菜用到的每种食材都是这道菜的一个属性。

要计算两个由复合成员组成的集合的交集，必须查找所有属性都匹配的成员；另外，在这两个集合中，所有成员包含的属性个数和类型都必须相同。例如，假设有一个类似于下面的复集（complex set），其中每行都是一个集成员

（菜品），而每列都表示一个属性（食材）。

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Pasta	Water	Pork	Onions

另外，假设还有一个类似于下面的集合：

Potatoes	Water	Lamb	Onions
Rice	Chicken Stock	Turkey	Carrots
Pasta	Vegetable Stock	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Beans	Water	Pork	Onions

这两个集合的交集包含这两个集合中所有属性都匹配的成员，如下所示：

Potatoes	Beef Stock	Beef	Cabbage

2. 结果集的交集

如果你觉得前面的示例像是表行或结果集，这种感觉完全正确。在使用 SQL 取回的数据集中的行中，各列都是属性。例如，假设你使用查询返回了一系列行，它们类似于下面这样（摘自我的一本烹饪书）：

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Chicken Stew	Rice	Chicken Stock	Chicken	Carrots
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Pasta	Water	Pork	Onions

假设还有一个查询结果集，它类似于下面这样（摘自我朋友 Mike 的一本烹饪书）：

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Beans	Water	Pork	Onions

这两个集合的交集包含两个属性都匹配的成员，即在我和 Mike 的烹饪书中都有的两个食谱：

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage

在有些情况下，使用集合图更容易说明白交集运算的工作原理。**集合图**是一种简单而优雅的方法，使用图形来表示集合以及它们的重叠情况。你可能听说过，这种图被称为欧拉图或韦恩图。顺便说一句，欧拉是 18 世纪的瑞士数学家，而韦恩在他于 1880 年在剑桥大学担任研究员时撰写的一篇论文中使用了这种逻辑图，由此可知“集合思维”并非什么新概念。

假设有一个数据库，其中包括你喜欢的所有菜品，而你很喜欢使用洋葱来改善牛肉的味道，因此想找出所有包含牛肉和洋葱的菜品。图 7-1 所示的集合图可帮助你弄明白如何解决这个问题。

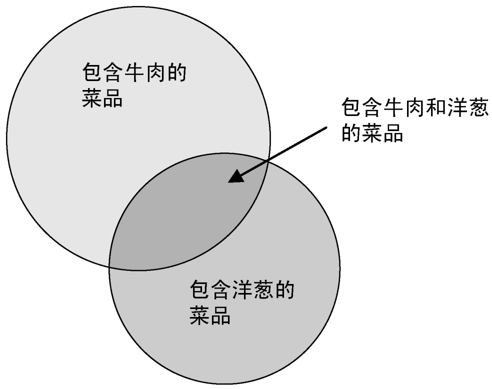


图 7-1　找出包含牛肉和洋葱的菜品

上面的圆表示由包含牛肉的菜品组成的集合，下面的圆表示由包含洋葱的菜品组成的集合。这两个圆的重叠部分是两个集合的交集，表示包含牛肉和洋葱的菜品。可以想见，你首先让 SQL 取回所有包含牛肉的菜品；在第二个查询中，你让 SQL 取回所有包含洋葱的菜品。后面你将看到，要得到最终的答案，可使用一个特殊的 SQL 关键字 INTERSECT 将这两个查询关联起来。

我知道你在想什么：如果菜品表像上面这样，只需像下面这样请求即可：

“列出将牛肉用作肉食食材并将洋葱作为蔬菜食材的菜品。”

转换	Select the recipe name from the recipes table where meat ingredient is beef and vegetable ingredient is onions（从菜品表中选择肉食食材为牛肉、蔬菜食材为洋葱的菜品的名称）
整理	Select ~~the~~ recipe name from ~~the~~ recipes ~~table~~ where meat ingredient ~~is~~ = 'beef' and vegetable ingredient ~~is~~ = 'onions'

SQL	`SELECT RecipeName`
	`FROM Recipes`
	`WHERE MeatIngredient = 'Beef'`
	`   AND VegetableIngredient = 'Onions'`

且慢！如果你还记得在第 2 章学到的经验教训，就知道只包含 Recipes 表可能不可行。如果菜品并非只包含肉食食材和蔬菜食材呢？如果有些菜品包含的食材很多，而有些只有几样呢？要正确地设计菜品数据库，必须在其中包含一个 Recipe_Ingredients 表，其中每行都是一个菜品和食材组合。这个表的每行都只包含一种食材，而不可能同时包含牛肉和洋葱。因此需要先找出所有包含牛肉的行，再找出所有包含洋葱的行，然后基于 RecipeID 计算它们的交集（如果你不明白我为何批评前面的表设计，请务必回过头去阅读第 2 章）。

如果问题更复杂呢？假设你要再加上一种食材——胡萝卜。图 7-2 所示的集合图可帮助你设计解决方案。

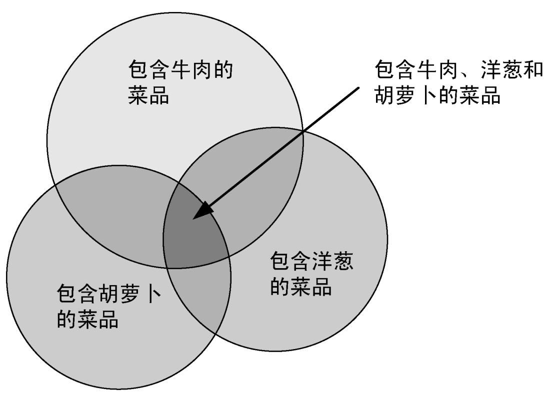


图 7-2　找出哪些菜品包含牛肉、洋葱和胡萝卜

明白了吗？总之，需要解决涉及复杂条件的问题时，集合图对你明白如何使用 SQL 结果集的交集来提供解决方案大有裨益。

3. 可使用交集来解决的问题

你可能猜到了，可使用交集来找出多个信息集的共同之处。下面列出了几个通过对示例数据库中的数据执行交集运算可解决的问题。

“列出姓名相同的顾客和员工。”
“找出订购了自行车和头盔的顾客。”
“列出为顾客 Bonnicksen 和 Rosales 都演出过的演唱组合。”
“列出艺术课程和计算机科学课程的平均分都不低于 85 的学生。”
“找出在保龄球馆 Thunderbird Lanes 和 Bolero Lanes 举行的比赛中原始得分都不低于 155 的投球手。”
“列出包含牛肉和大蒜的菜品。”

使用严格的交集运算时，一个限制是所有列的值都必须匹配。对来自同一个表的多个结果集（如订购了自行车的顾客和订购了头盔的顾客）执行交集运算时，这不会有任何问题；对来自包含类似列的表的结果集（如顾客姓名和员工姓名）执行交集运算时，也如此。然而，在很多情况下，需要解决的问题都只要求几列的值匹配。对于这种问题，SQL 提供了 JOIN 操作——基于关键值的交集运算。下面是一些可使用 JOIN 来解决的问题。

“列出居住在同一座城市的顾客和员工（基于城市名的 JOIN）。”
“列出顾客及其签约的演唱组合（基于演出编号的 JOIN）。”
“找出居住地点的邮政编码相同的经纪人和演唱组合（基于邮政编码的 JOIN）。”
“列出名字相同的学生和老师（基于名字的 JOIN）。”
“找出属于同一个球队的投球手（基于球队 ID 的 JOIN）。”
“列出使用了胡萝卜的菜品使用的所有食材（基于食材 ID 的 JOIN）。”

别担心，下一章将介绍如何使用 JOIN 解决所有这些问题（还有其他的问题）。鉴于几乎没有商业实现支持 SQL 标准定义的 INTERSECT，我将介绍如何使用 JOIN 来解决原本需要使用 INTERSECT 的问题。

7.2.2　差集

21 和 10 的差是多少呢？如果你回答 11，那就答对了！**差集**运算也叫剔除、减去或排除（except）运算，它从一个集合中删除出现在第二个集合中的值，只余下第一个集合中未出现在第二个集合中的值（后面你将看到，SQL 标准使用的相应关键字为 EXCEPT）。

1. 集合论中的差集

差集运算是另一个很强大的数学工具。科学家可能想知道两个化学或物理样本数据有何不同。例如，从事药剂研究的化学家可能发现两种药剂很像，但一种有效，另一种无效，而找出这两种药剂的差别可能有助于揭示其中的原因。工程师可能有两个相同的设计，但一个设计的效果比另一个好，找出这两种设计的差异可能对消除未来建筑的结构缺陷至关重要。

下面来看看如何计算两个数字集合的差集。第一个数字集合如下：

1, 5, 8, 9, 32, 55, 78

第二个数字集合如下：

3, 7, 8, 22, 55, 71, 99

这两个数字集合的差集包含出现在第一个集合中但未出现在第二个集合中的数字：

1, 5, 9, 32, 78

请注意，可将前面的差集运算反转过来，计算第二个集合与第一个集合的差集，结果如下：

3, 7, 22, 71, 99

集合的成员并非只能包含一个值。实际上，使用 SQL 解决问题时，涉及的很可能是由行组成的集合。

本章前面说过，集合成员包含多个值时，就有多个属性（描述成员特征的信息）。例如，你喜欢的菜品就是由所有菜品组成的集合中的复合成员，包含很多不同的食材，而你可将每种食材视为这种复合成员的属性。

要计算两个复集成员的差集，必须找出两个集合中所有属性都匹配的成员（别忘了，两个集合的成员包含的属性个数和类型必须相同），再从第一个集合中减去这些成员，便可得到差集。例如，假设有一个类似于下面的复集，其中每行都表示一个集合成员（菜品），而每列都表示一个属性（食材）：

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Pasta	Water	Pork	Onions

假设还有一个集合，它类似于下面这样：

Potatoes	Water	Lamb	Onions
Rice	Chicken Stock	Turkey	Carrots
Pasta	Vegetable Stock	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Beans	Water	Pork	Onions

这两个集合的差集包含出现在第一个集合中但未出现在第二个集合中的元素：

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Pasta	Water	Pork	Onions

2. 结果集的差集

在使用 SQL 取回的数据集的行中，各列都是属性。例如，假设你使用查询返回了一系列行，它们类似于下面这样（摘自我的一本烹饪书）：

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Chicken Stew	Rice	Chicken Stock	Chicken	Carrots
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Pasta	Water	Pork	Onions

假设还有一个查询结果集，它类似于下面这样（摘自我朋友 Mike 的一本烹饪书）：

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Beans	Water	Pork	Onions

我的菜品和 Mike 的菜品的差集包含出现在我的烹饪书中但未出现在 Mike 的烹饪书中的菜品。

Recipe	Starch	Stock	Meat	Vegetable
Chicken Stew	Rice	Chicken Stock	Chicken	Carrots
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Pork Stew	Pasta	Water	Pork	Onions

也可将这个问题反过来：查找出现在 Mike 的烹饪书中但未出现我的烹饪书中的菜品。结果如下：

Recipe	Starch	Stock	Meat	Vegetable
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Pork Stew	Beans	Water	Pork	Onions

同样，可使用集合图来帮助理解差集运算的工作原理。假设有一个数据库，其中包括你喜欢的所有菜品，而你很不喜欢洋葱对牛肉味道的影响，因此想找出所有包含牛肉但不包含洋葱的菜品。图 7-3 所示的集合图可帮助你搞明白如何解决这个问题。

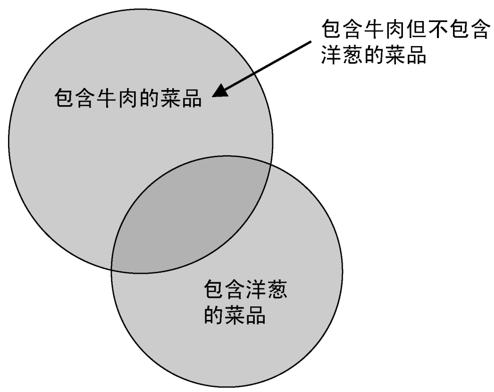


图 7-3　找出哪些菜品包含牛肉但不包含洋葱

上面的圆表示由包含牛肉的菜品组成的集合，下面的圆表示由包含洋葱的菜品组成的集合。前面讨论交集时说过，这两个圆重叠的部分就是包含牛肉和洋葱的菜品。在上面那个圆中，除重叠区域外的部分表示包含牛肉但不包含洋葱的菜品；同理，在下面那个圆中，除重叠区域外的部分表示包含洋葱但不包含牛肉的菜品。

你现在可能知道了，需要先让 SQL 取回所有包含牛肉的菜品，再让 SQL 取回所有包含洋葱的菜品。在本章后面你将看到，要得到最终的答案，可使用特殊的 SQL 关键字 EXCEPT 来关联这两个查询。

你会再次跌入前面所说的陷阱吗（你阅读过第 2 章吗）？换而言之，你可能认为，如果菜品表像上面那样，发出下面这样的请求即可：

“列出将牛肉用作肉食食材但不将洋葱作为蔬菜食材的菜品。”

转换	Select the recipe name from the recipes table where meat ingredient is beef and vegetable ingredient is not onions（从菜品表中选择肉食食材为牛肉且蔬菜食材不为洋葱的菜品的名称）
整理	Select ~~the~~ recipe name from ~~the~~ recipes ~~table~~ where meat ingredient ~~is~~ = 'beef' and vegetable ingredient ~~is not~~ <> 'onions'

SQL	`SELECT RecipeName`
	`FROM Recipes`
	`WHERE MeatIngredient = 'Beef'`
	`   AND VegetableIngredient <> 'Onions'`

再次重申，第 2 章说过，只包含 Recipes 表可不是好主意。如果菜品并非只包含肉食食材和蔬菜食材呢？如果有些菜品包含的食材很多，而有些只有几样呢？要正确地设计菜品数据库，必须在其中包含一个 Recipe_Ingredients 表，其中每行都是一个菜品和食材组合。这个表的每行都只包含一种食材，而不可能同时包含牛肉和洋葱。因此需要先找

出所有包含牛肉的行，再找出所有包含洋葱的行，然后基于 RecipeID 计算它们的差集。

如果问题更复杂呢？假设你也讨厌胡萝卜，图 7-4 所示的集合图可帮助你设计解决方案。

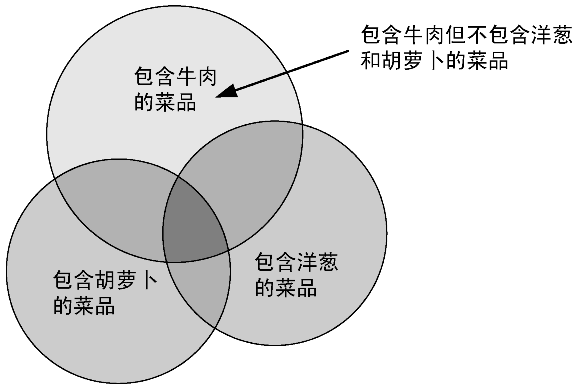


图 7-4　找出哪些菜品包含牛肉但不包含洋葱和胡萝卜

你首先需要获得由包含牛肉的菜品组成的集合，再计算这个集合与由包含洋葱的菜品组成的集合（或由包含胡萝卜的菜品组成的集合）的差集，然后计算结果与由包含胡萝卜的菜品组成的集合（或由包含洋葱的菜品组成的集合）的差集，结果便是包含牛肉但不包含胡萝卜或洋葱的菜品（上面那个圆中的浅色区域）。

3. 可使用差集运算来解决的问题

不像交集运算那样找出两个集合都有的成员，差集运算找出出现在一个集合中但未出现在另一个集合中的成员。下面列出了几个通过对示例数据库中的数据执行差集运算可解决的问题。

“列出姓名不与任何员工相同的顾客。”

“找出所有订购了自行车但没有订购头盔的顾客。”

“列出为顾客 Bonnicksen 演出给但没有为顾客 Rosales 演出过的演唱组合。”

“列出艺术课程平均成绩不低于 85 但计算机科学平均成绩低于 85 的学生。”

“找出在 Thunderbird Lanes 保龄球馆比赛时原始得分不低于 155，但在 Bolero Lanes 保龄球馆比赛时原始得分低于 155 的投球手。”

“列出包含牛肉但不包含大蒜的菜品。”

使用严格的差集运算时，一个限制是所有列的值都必须匹配。对来自同一个表的多个结果集（如订购了自行车的顾客和订购了头盔的顾客）执行差集运算时，这不会有任何问题；对来自包含类似列的表的结果集（如顾客姓名和员工姓名）执行差集运算时，也如此。

然而，在很多情况下，需要解决的问题都只要求几列的值匹配。对于这种问题，SQL 提供了 OUTER JOIN 操作：基于关键值的交集运算（包括一个或两个集合中不匹配的元素）。下面是一些可使用 OUTER JOIN 来解决的问题。

“列出不与任何员工居住在同一座城市的顾客（基于城市名的 OUTER JOIN）。”

“列出顾客及其没有与之签约的演唱组合（基于演出合约编号的 OUTER JOIN）。”

“找出居住地点的邮政编码不与任何演唱组合相同的经纪人（基于邮政编码的 OUTER JOIN）。”

“列出名字不与任何老师相同的学生（基于名字的 OUTER JOIN）。”

“找出平均得分不低于 150 且得分从未低于 125 的投球手（基于两个表中投球手 ID 的 OUTER JOIN）。”

“列出没有使用胡萝卜的菜品使用的所有食材（基于菜品 ID 的 OUTER JOIN）。”

别担心，第 9 章将介绍如何使用 OUTER JOIN 解决所有这些问题（还有其他问题）。另外，由于只有几种商业实现支持 SQL 标准定义的 EXCEPT（表示差集运算的关键字），我将介绍如何使用 OUTER JOIN 来解决原本需要使用 EXCEPT 的问题，而第 18 章将介绍解决这类问题的其他方式。

7.2.3　并集

前面介绍了如何查找两个集合中都有的元素（交集）以及两个集合中不同的元素（差集）。第三种集合运算是将两个集合相加（并集）。

1. 集合论中的并集

并集运算让你能够将两个包含类似信息的集合合并成一个。科学家可能想合并两个化学或物理样本数据集。例如，从事药剂研究的化学家可能有两组有特定效果的药剂，通过合并这两个集合，可获得一种具有全效的药剂。

下面来看看如何计算两个数字集合的并集。第一个数字集合如下：

1, 5, 8, 9, 32, 55, 78

第二个数字集合如下：

3, 7, 8, 22, 55, 71, 99

对这两个数字集合执行并集运算时，将把其中的数字合并成一个新集合：

1, 5, 8, 9, 32, 55, 78, 3, 7, 22, 71, 99

请注意，在结果中这两个集合中都有的数字（8 和 55）只出现了一次；另外，结果中数字的排列顺序是不确定的。当你让数据库系统执行并集运算时，除非指定了 ORDER BY 子句，否则结果集中值的排列顺序将是不确定的。在 SQL 中，如果要保留重复的数字，可使用 UNION ALL。

集合的成员并非只能包含一个值。实际上，使用 SQL 解决问题时，涉及的很可能是由行组成的集合。

要计算两个复集的并集，它们的成员包含的属性个数和类型必须相同。例如，假设有一个类似于下面的复集，其中每行都表示一个集合成员（菜品），而每列都表示一个属性（食材）：

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Pasta	Water	Pork	Onions

假设还有一个集合，它类似于下面这样：

Potatoes	Water	Lamb	Onions
Rice	Chicken Stock	Turkey	Carrots
Pasta	Vegetable Stock	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Beans	Water	Pork	Onions

这两个集合的并集包含这两个集合中的所有元素（删除重复的）：

Potatoes	Water	Lamb	Peas
Rice	Chicken Stock	Chicken	Carrots
Pasta	Water	Tofu	Snap Peas
Potatoes	Beef Stock	Beef	Cabbage
Pasta	Water	Pork	Onions
Potatoes	Water	Lamb	Onions
Rice	Chicken Stock	Turkey	Carrots
Pasta	Vegetable Stock	Tofu	Snap Peas
Beans	Water	Pork	Onions

2. 使用并集运算合并结果集

复集中的元素与 SQL 结果集中的行很像，在使用 SQL 取回的数据集的行中，每列都是属性。例如，假设你使用查询返回了一系列行，它们类似于下面这样（摘自我的一本烹饪书）：

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Chicken Stew	Rice	Chicken Stock	Chicken	Carrots
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Pasta	Water	Pork	Onions

假设还有一个查询结果集，它类似于下面这样（摘自我朋友 Mike 的一本烹饪书）：

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Beans	Water	Pork	Onions

这两个集合的并集包含这两个集合中的所有行（也许我和 Mike 决定合著一本烹饪书）：

Recipe	Starch	Stock	Meat	Vegetable
Lamb Stew	Potatoes	Water	Lamb	Peas
Chicken Stew	Rice	Chicken Stock	Chicken	Cabbage
Veggie Stew	Pasta	Water	Tofu	Snap Peas
Irish Stew	Potatoes	Beef Stock	Beef	Cabbage
Pork Stew	Pasta	Water	Pork	Onions
Turkey Stew	Rice	Chicken Stock	Turkey	Carrots
Veggie Stew	Pasta	Vegetable Stock	Tofu	Snap Peas
Pork Stew	Beans	Water	Pork	Onions

假设有一个数据库，其中包括你喜欢的所有菜品，而你很喜欢包含洋葱或牛肉的菜品，因此想找出所有包含这两种食材之一的菜品。图 7-5 所示的集合图可帮助你搞明白如何解决这个问题。

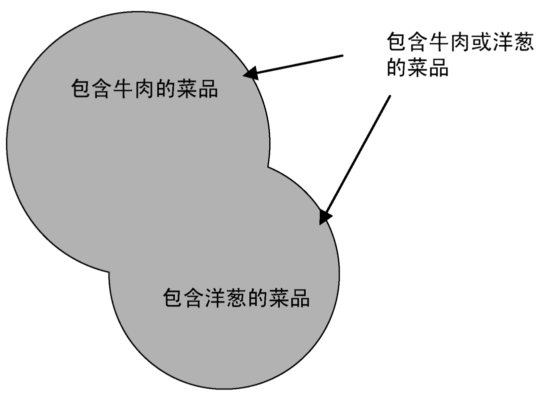


图 7-5　找出哪些菜品包含牛肉或洋葱

上面的圆表示由包含牛肉的菜品组成的集合，下面的圆表示由包含洋葱的菜品组成的集合。这两个圆一起表示所

有包含这两种食材之一的菜品（删除了两个集合重叠处的重复部分）。你可能知道，需要先让 SQL 取回所有包含牛肉的菜品，再在第二个查询中让 SQL 取回所有包含洋葱的菜品。你在后面将看到，要获得最终的答案，可使用 SQL 关键字 UNION 将这两个查询关联起来。

你知道，将菜品数据库设计成只包含一个表并非好主意。相反，正确的设计是在其中包含一个 Recipe_Ingredients 表，其中每行都是一个菜品和食材组合。这个表的每行都只包含一种食材，而不可能同时包含牛肉和洋葱。因此需要先找出所有包含牛肉的行，再找出所有包含洋葱的行，然后计算它们的并集。

3. 可使用并集运算解决的问题

并集运算让你能够将来自两个类似集合的行合并，并消除重复的行。下面列出了几个通过对示例数据库中的数据执行并集运算可解决的问题。

“列出所有顾客和员工的姓名和地址。”

“列出订购了自行车的顾客以及订购了头盔的顾客。”

“列出为顾客 Bonnicksen 演出过的演唱组合以及为顾客 Rosales 演出过的演唱组合。”

“列出艺术课程的平均成绩不低于 85 的学生，以及计算机科学的平均成绩不低于 85 的学生。”

“找出在保龄球馆 Thunderbird Lanes 比赛时原始得分不低于 155 的投球手，以及在保龄球馆 Bolero Lanes 比赛时原始得分不低于 140 的投球手。”

“列出包含牛肉的菜品以及包含大蒜的菜品。”

与其他严格的集合运算一样，一个限制是所有列的值都必须匹配。对来自同一个表的多个结果集（如订购了自行车的顾客和订购了头盔的顾客）执行并集运算时，这不会有任何问题；对来自包含类似列的表的结果集（如顾客的姓名、地址和员工的姓名、地址）执行差集运算时，也如此。第 10 章将详细介绍 SQL 运算符 UNION 的用法。

在通过使用并集运算合并来自同一个表的行可解决的问题中，很多也可使用 DISTINCT（消除重复的行）并基于复杂的条件对表进行连接来解决，第 8 章将详细介绍这种使用 JOIN 解决问题的方式。

7.3　SQL 集合运算

对集合运算有了基本认识后，来看看它们在 SQL 中是如何实现的。

7.3.1　经典集合运算和 SQL

前面说过，直接支持并集运算（INTERSECT）和差集运算的（EXCEPT）商业数据库系统不多。然而，SQL 标准明确地规定了该如何实现这些运算。考虑到这些集合运算很重要，有必要简单地介绍一下它们的语法。

前面承诺过，本书后面将介绍使用 JOIN 解决交集和差集问题的方式。由于大多数数据库系统支持 UNION，因此第 10 章将专门介绍其用法。本章余下的篇幅将概述这三种运算。

7.3.2　查找都有的值：INTERSECT

假设你要解决如下看起来很简单的问题：

“列出包含自行车和头盔的订单。”

转换	Select the distinct order numbers from the order details table where the product number is in the list of bike and helmet product numbers（从订单详情表中选择商品编号在自行车和头盔编号列表中的不同订单编号）
整理	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike and helmet product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN`
	`   (1, 2, 6, 10, 11, 25, 26)`

❖ **说明**　熟悉 SQL 的读者可能会问：为何不连接 Order_Details 和 Products 表，并查找商品名“自行车”（bike）和“头盔”（helmet）呢？简单地说，是因为还没有介绍连接的概念，因此这里查询单个表，并使用谓词 IN 和已知的自行车和头盔商品编号列表。

乍一看这可行，但包含自行车或头盔的订单也包含在结果中，而你想要找出的是包含自行车和头盔的订单！如果将包含自行车的订单和包含头盔的订单视为两个不同的集合，这个问题将更容易理解。图 7-6 使用集合图说明了这两个集合之间的可能关系。

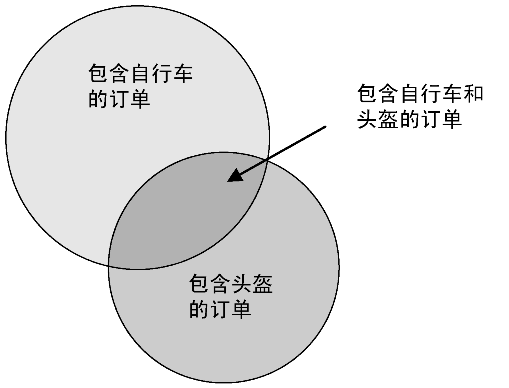


图 7-6　两个订单集合之间的可能关系

实际上，无法预测两个数据集之间是什么关系。在图 7-6 中，有些订单的商品列表中包含自行车，但不包含头盔；有些包含头盔，但不包含自行车。在这两个集合的重叠区域（即交集）中，是包含自行车和头盔的订单。图 7-7 说明了另一种情形，即包含头盔的订单都包含自行车，但有些订单包含自行车而不包含头盔。

请求中的词语“同时”可能意味着需要生成两个不同的数据集，再以某种方式将它们组合起来（这里的请求就是这样的）。

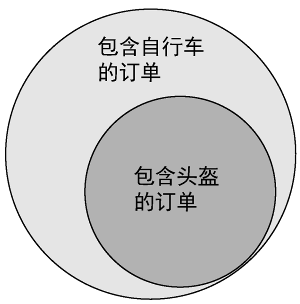


图 7-7　包含头盔的订单都包含自行车

“列出包含自行车的订单。”

转换	Select the distinct order numbers from the order details table where the product number is in the list of bike product numbers（从订单详情表中选择商品号在自行车商品号列表中的不同订单号）
整理	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN (1, 2, 6, 11)`

“列出包含头盔的订单。”

转换	Select the distinct order numbers from the order details table where the product number is in the list of helmet product numbers（从订单详情表中选择商品号在头盔商品号列表中的不同订单号）
整理	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ helmet product numbers
SQL	SELECT DISTINCT OrderNumber FROM Order_Details WHERE ProductNumber IN (10, 25, 26)

现在可以对这两个集合执行**交集**运算，得到最终的解决方案了。图 7-8 显示了处理这个问题的 SQL 语法图（请注意，可多次使用 INTERSECT 来合并多条 SELECT 语句）。

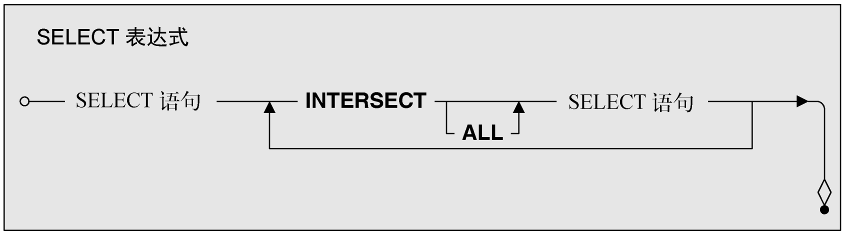


图 7-8　使用 INTERSECT 关联两条 SELECT 语句

现在可以使用运算符 INTERSECT 将请求的两部分关联起来，获得正确的答案了：

SQL	SELECT DISTINCT OrderNumber FROM Order_Details WHERE ProductNumber IN (1, 2, 6, 11) INTERSECT SELECT DISTINCT OrderNumber FROM Order_Details WHERE ProductNumber IN (10, 25, 26)

遗憾的是支持运算符 INTERSECT 的商业 SQL 实现不多，但也是有的！别忘了，表的主键唯一地标识了每一行（要找出交集中的行，不要求所有列都匹配，而只需主键匹配）。第 8 章将介绍另一种解决这种问题的方法——使用连接（JOIN）。好消息是几乎所有商业 SQL 实现都支持连接。

7.3.3　查找没有的值：EXCEPT（差集）

回到前面有关自行车和头盔的问题。假设你要应答下面这个看似简单的请求：

“列出包含自行车但不包含头盔的订单。”

转换	Select the distinct order numbers from the order details table where the product number is in the list of bike product numbers and product number is not in the list of helmet product numbers（从订单详情表中选择商品号在自行车商品号列表中，但不在头盔商品号列表中的不同订单号）
整理	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike product numbers and product number ~~is~~ not in ~~the list of~~ helmet product numbers
SQL	SELECT DISTINCT OrderNumber FROM Order_Details WHERE ProductNumber IN (1, 2, 6, 11) AND ProductNumber NOT IN (10, 25, 26)

不幸的是，这列出的是只包含自行车的订单。问题出在第一个 IN 子句查找包含自行车的订单详情行，而第二个 IN 子句删除包含头盔的行。如果将包含自行车的订单和包含头盔的订单视为两个不同的集合，这个请求将更容易理解。图 7-9 说明了这两个订单集合之间的一种可能关系。

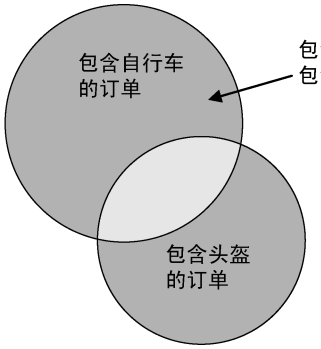


图 7-9 包含自行车但不包含头盔的订单

请求中的词语“除外”或“但不”意味着可能必须先获取两个不同的数据集，再以某种方式组合它们（这里的请求就是这样的）。

“列出包含自行车的订单。”

转换	Select the distinct order numbers from the order details table where the product number is in the list of bike product numbers（从订单详情表中选择商品号在自行车商品号列表中的不同订单号）
整理	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN (1, 2, 6, 11)`

“列出包含头盔的订单。”

转换	Select the distinct order numbers from the order details table where the product number is in the list of helmet product numbers（从订单详情表中选择商品号在头盔商品号列表中的不同订单号）
整理	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ helmet product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN (10, 25, 26)`

现在可以对这两个集合执行差集运算来获得最终的解决方案了。SQL 使用关键字 EXCEPT 表示差集运算。图 7-10 展示了处理这种问题的 SQL 语法图。

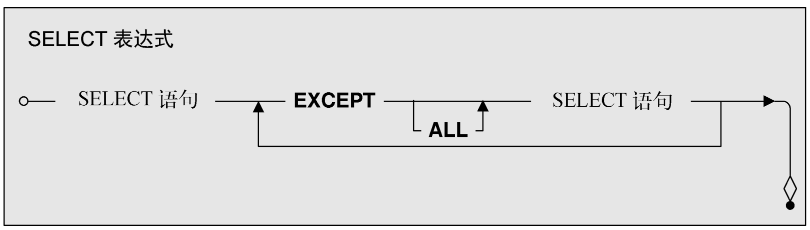


图 7-10 使用 EXCEPT 将两条 SELECT 语句关联起来

现在可以使用运算符 EXCEPT 将请求的两部分关联起来，获得正确的答案了：

SQL

```
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (1, 2, 6, 11)
EXCEPT
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (10, 25, 26)
```

前面讨论差集运算时说过，集合的顺序很重要。这里要查找的是包含自行车但不包含头盔的订单；如果相反，即要查找包含头盔但不包含自行车的订单，可反转过来，如下所示：

SQL

```
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (10, 25, 26)
EXCEPT
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (1, 2, 6, 11)
```

遗憾的是，支持 EXCEPT 运算符的商业 SQL 实现不多。别忘了，表的主键唯一地标识了每一行（要找出差集中的行，不要求所有列都匹配，而只需主键匹配）。第 9 章将介绍另一种解决这类问题的方法——使用外连接（OUTER JOIN）。好消息是几乎所有商业 SQL 实现都支持外连接。

7.3.4　合并集合：UNION

结束本章前，再介绍一个有关自行车和头盔的问题。假设你要应答下面这个表面上看起来很简单的请求：

“列出包含自行车或头盔的订单。”

转换	Select the distinct order numbers from the order details table where the product number is in the list of bike and helmet product numbers（从订单详情表中选择商品号在自行车商品号和头盔商品号列表中的不同订单号）
整理	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike and helmet product numbers

SQL

```
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (1, 2, 6, 10, 11, 25, 26)
```

实际上，这确实可行！既然如此，为何要使用 UNION 来解决这个问题呢？实际上，你可能不会使用 UNION，但如果问题更复杂，UNION 将很有用：

“列出订购了自行车的顾客以及提供自行车的供应商。”

要应答这个请求，需要使用连接运算创建两个查询，再使用 UNION 获得最终结果。鉴于还没有介绍如何执行连接，因此我把这个问题留到第 10 章再解决。这是不是让你心里有所期待？

回到前面“列出包含自行车或头盔的订单”的问题，并使用 UNION 来解决它。如果将包含自行车的订单和包含头盔的订单视为两个不同的集合，你将发现这个问题更容易理解。图 7-11 显示了这两个集合之间的可能关系。

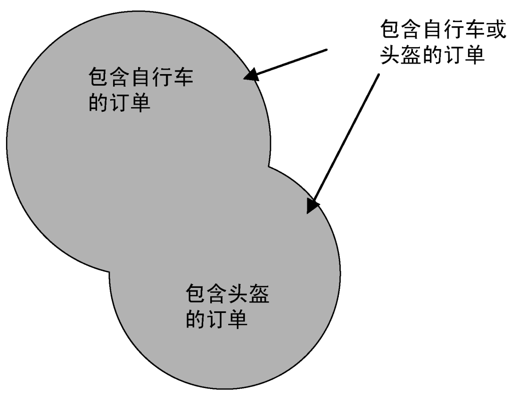


图 7-11　包含自行车或头盔的订单

请求中的词语“或”“还有”可能意味着需要获得两个不同的数据集，再使用 UNION 组合它们。这个请求可分为两部分：

“列出包含自行车的订单。”

转换	Select the distinct order numbers from the order details table where the product number is in the list of bike product numbers（从订单详情表中选择商品号在自行车商品号列表中的不同订单号）
整理	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ bike product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN (1, 2, 6, 11)`

“列出包含头盔的订单。”

转换	Select the distinct order numbers from the order details table where the product number is in the list of helmet product numbers（从订单详情表中选择商品号在头盔商品号列表中的不同订单号）
整理	Select ~~the~~ distinct order numbers from ~~the~~ order details ~~table~~ where ~~the~~ product number ~~is~~ in ~~the list of~~ helmet product numbers

SQL	`SELECT DISTINCT OrderNumber`
	`FROM Order_Details`
	`WHERE ProductNumber IN (10, 25, 26)`

现在可以对这两个集合执行**并集**运算来获得最终的解决方案了。图 7-12 展示了处理这个问题的语法图。

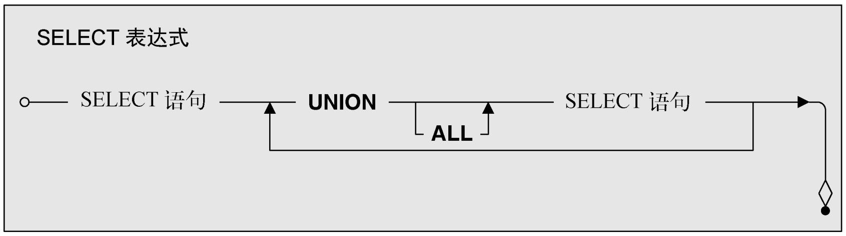


图 7-12　使用 UNION 关联两条 SELECT 语句

现在可以使用 UNION 运算符将请求的两部分关联起来，获得正确的答案了：

SQL

```
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (1, 2, 6, 11)
UNION
SELECT DISTINCT OrderNumber
FROM Order_Details
WHERE ProductNumber IN (10, 25, 26)
```

好消息是几乎所有商业 SQL 实现都支持 UNION 运算符。从这些示例可知，要从单个表中获得“或”型结果时，使用 UNION 可能比较麻烦。UNION 最大的用武之地是使用多个结构类似的表来生成列表，这将在第 10 章更详细地探讨。

7.4　小结

本章首先讨论了集合的概念。接下来详细讨论了 SQL 实现的每种主要集合运算——交集、差集和并集，并介绍了如何使用集合图来帮助理解要解决的问题。最后，简要地介绍了与这三种运算相关的 SQL 语法和关键字（INTERSECT、EXCEPT 和 UNION），这旨在吊起你的胃口。

此时你可能会说，且慢，你介绍了三种集合运算，但其中的两种我可能根本不会用到，这是为什么呢？别忘了本章的标题是“集合思维”。但凡要解决复杂问题，就需要将问题分成多个结果集，再将它们组合起来。

因此，如果问题涉及“必须这样且必须那样”，可能需要分别解决“这样”和“那样”，再将它们关联起来获得最终的解决方案。SQL 标准定义了方便的 INTERSECT 操作，但对于使用 INTERSECT 能解决的问题，也可使用 INNER JOIN 来解决，这将在第 8 章介绍。

同理，如果问题涉及“必须这样但不能那样”，可能需要分别解决“这样”和“那样”，再从“这样”减去“那样”来得到答案。本章介绍了 SQL 标准操作 EXCEPT，但对于使用 EXCEPT 能解决的问题，也可使用 OUTER JOIN 来解决，详情请参阅第 9 章和第 18 章。

最后，本章演示了如何使用 UNION 来将信息集相加，但有关它的详细介绍要等到第 10 章。

第8章 内 连 接

“不要让灵感和想象力熄灭；不要成为模特的奴隶。”

——文森特·梵高

本章涵盖如下主题：

- 何谓连接
- 内连接
- 内连接的用途
- 语句举例
- 小结
- 练习

本书前面介绍的都是如何使用单个表来解决问题。你现在知道如何从一个表中获得简单答案，还知道如何通过使用表达式或对结果集进行排序来获得更复杂的答案。换而言之，你能画出惟妙惟肖的眼睛、下巴、嘴巴和鼻子，而本章将介绍如何将多个部分组合起来，画出一幅肖像。

8.1 何谓连接

第2章强调了将表中数据分成多个客体的重要性,但在现实世界中你面临的大多数问题要求你组合多个表的数据：顾客及其订单、顾客及其签约的演唱组合、投球手及其得分、学生及其注册的课程、菜品及制作它所需的食材。要解决这些更复杂的问题，必须连接多个表才能找到答案，为此可使用关键字 JOIN。

前一章演示了计算两个数据集的交集对解决问题很有帮助，但你可能还记得，INTERSECT 要求两个结果集中的列都匹配。连接也是一种交集运算，但与 INTERSECT 不同，因为它让数据库系统基于指定的列来执行交集运算。因此，连接让你能够基于共有的列计算两个截然不同的表的交集，例如，可使用连接通过匹配 Customers 表中的 CustomerID 和 Orders 表中的 CustomerID，将顾客关联到其订单。

后面你将看到，连接是在 SQL 语句的 FROM 子句中指定的。连接定义了一个逻辑表，其中包含连接两个表或结果集的结果。通过在 FROM 子句中指定连接，定义了一个逻辑表，查询将从中提取最终的结果集。换而言之，在 FROM 子句中，可使用连接来替换本书前面使用的单个表名。本章后面你将看到，还可使用多个连接操作基于多个表创建复杂的结果集。

8.2 内连接

SQL 标准定义了多种执行连接的方式，其中最常用的是**内连接**（INNER JOIN）。假设要将学生关联到其注册的课程，但有些学生可能没有注册任何课程，同时有些课程没有任何学生注册。

将 Students 表内连接到 Classes 表时，将返回 Students 表中这样的行，即被关联到（通过 Student_Schedules 表）Classes 表中相关的行。然而，这不会返回没有注册任何课程的学生，也不会返回没有任何学生注册的课程。内连接只返回这样的行，即在两个表或结果集中用于建立连接的列的值匹配。

8.2.1　可基于什么进行连接

大多数情况下，基于一个表的主键和另一个表的相关外键来建立连接。第 2 章说过，外键的数据类型必须与相关的主键相同。然而，连接两个表或结果集时，也可基于任何满足如下条件的列：其数据类型为 SQL 标准定义的“可连接”数据类型。

一般而言，可将字符列连接到字符列或字符表达式，可将数字列（如整数列）连接到任何数字列（可能是浮点数列），可将日期列连接到任何日期列。因此，可基于城市列或邮政编码列将 Customers 表连接到 Employees 表（这可能旨在找出哪些顾客和员工居住在相同的城市或邮区）。

> ❖ **说明**　可基于两个表中任何“可连接”列来定义连接并不意味着应该这样做。用于建立连接的列的数据类型必须相同，这样连接才有意义。例如，即便顾客姓名和员工地址都是字符数据类型，但将它们连接起来毫无意义。如果你这样做，结果集中将不会有任何行，除非有人在员工地址中错误地输入了姓名。同理，将 StudentID 连接到 ClassID 也毫无意义，虽然它们都是数字。如果你这样做，结果集中可能包含一些行，但没有任何意义。
>
> 即便连接有意义，编写出的查询也可能需要很长时间才能执行完毕。例如，如果数据库管理员没有为用于建立连接的列定义索引，数据库系统可能需要做大量额外的工作。另外，如果基于表达式（如拼接两个表中名和姓的表达式）来建立连接，数据库系统将不仅需要在所有行中根据表达式来生成一列，还可能必须多次扫描两个表中的所有数据，以便返回正确的结果。

8.2.2　列引用

讨论连接的语法前，再介绍一个要点。本书前面创建的查询都是基于单个表的，因此无须对列名进行限定。但创建涉及多个表的查询（使用连接时就如此）时，这些表常常包含名称相同的列。在第 2 章，我建议通过复制主键（包括其名称）在相关表中创建外键。

那么，如何明确地告诉数据库系统你在查询语法中指的是哪列呢？简单的答案是在**列引用**中包含表名。图 8-1 显示了列引用的语法图。

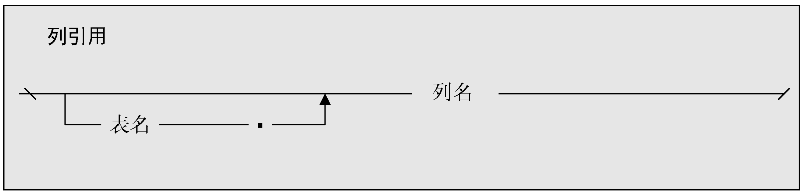


图 8-1　列引用的语法图

在使用 SQL 编写的语句的任何子句中，都可只使用列名本身，但也可使用表名来限定列名。如果列名在 FROM 子句包含的表中不是唯一的，就必须使用表名对其进行限定。下面演示了如何在查询 Employees 表的 SELECT 语句中使用限定的列名：

```
SQL        SELECT Employees.FirstName, Employees.LastName,
               Employees.PhoneNumber
           FROM Employees
```

说完这个注意事项后，可接着介绍连接操作的语法了。

8.2.3　语法

可将本书前面介绍的内容视为驾车经乡村公路或穿越小镇去买点杂货。下面系上安全带到高速公路上探险——探索内连接的语法。

1. 使用表

先来点简单的——基于两个表的内连接。图 8-2 说明了创建这种查询的语法。

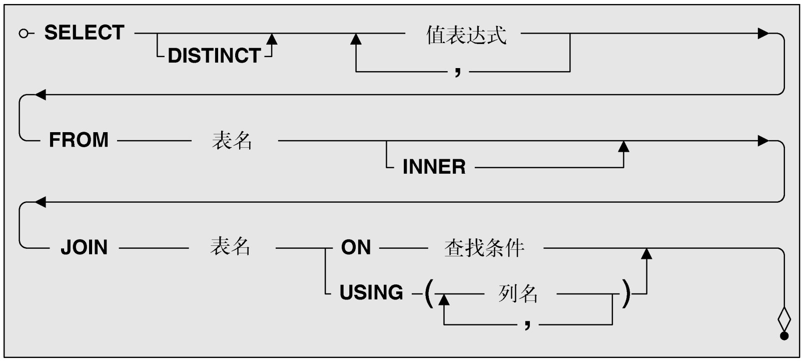


图 8-2　基于两个表的内连接查询的语法图

如你所见，现在 FROM 子句更复杂些（出于简化考虑，暂时省略了 WHERE 和 ORDER BY 子句）：不再只指定一个表名，而是指定两个表名并使用关联字 JOIN 将它们关联起来。请注意，关键字 INNER 指定连接的类型，是可选的。下一章将介绍，还可指定外连接（OUTER JOIN）。如果没有明确地指定连接的类型，将默认为内连接。建议总是明确地指定连接的类型，让请求更清晰。

❖ **说明**　如果你查看附录 A 中完整的语法图，将发现“表引用 JOIN 表引用”包含在术语**连接表**（Joined Table）中。**表引用**可以是表名，也可以是连接表，因此在 SELECT 语句的 FROM 子句中，使用的是表引用。为方便研究简单的两表连接，我在图中隐藏了这些复杂的定义。在本章后面的所有示意图中，我都将做这样的简化。

内连接的关键部分是第二个表名后面的 ON 或 USING 子句，它告诉数据库系统如何连接。为找出连接的结果，数据库系统在逻辑上将第一个表中的每行都与第二个表中的每行合并（这种合并被称为**笛卡儿积**，第 20 章将演示如何使用它来解决问题），再根据 ON 或 USING 子句指定的准则筛选出要返回的行。

第 6 章介绍了如何在 WHERE 子句中使用**查找条件**。在 ON 子句中，可使用查找条件来指定逻辑测试；对于给定的关联行，仅当满足该测试时才返回它们。编写查找条件时，必须至少将第一个表的一列与第二个表的一列进行比较，这样才有意义。虽然可编写非常复杂的查找条件，但通常只是比较一个表的主键与另一个表的外键是否相等。

来看一个简单的示例。在设计良好的数据库中，应将复杂的类别名放在另一个表中，并通过简单的键值将这些名称关联到主表。这样做有助于避免数据输入错误，因为这样数据库用户将选择类别名，而不是在每行中输入类别名（输入可能出现拼写错误）。例如，在示例数据库 Recipes 中，将菜品类型和菜品放在了不同的表中。图 8-3 说明了 Recipe_Classes 表和 Recipes 表之间的关系。

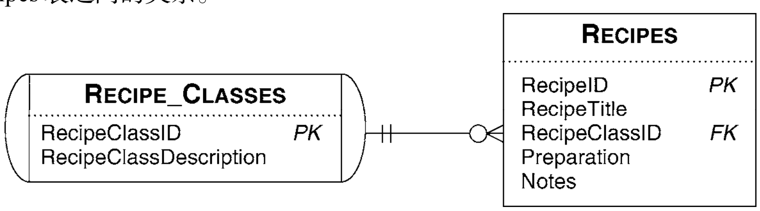


图 8-3　菜品类型描述放在一个独立于 Recipes 的表中

从数据库中检索有关菜品的信息以及菜品类型描述时，你不想看到 Recipes 表中的菜品类型 ID。来看看如何使用连接解决这个问题。

❖ **说明**　贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”方法。

“列出数据库中所有菜品的名称、制作方法和菜品类型描述。”

转换	Select recipe title, preparation, and recipe class description from the recipe classes table joined with the recipes table on recipe class ID in the recipe classes table matching recipe class ID in the recipes table（基于菜品类型 ID 相等连接菜品类型表和菜品表，并选择菜品名、制作方法和菜品类型描述）
整理	Select recipe title, preparation, ~~and~~ recipe class description from ~~the~~ recipe classes ~~table~~ inner join~~ed with the~~ recipes ~~table~~ on recipe_classes.recipe class ID ~~in the recipe classes table matching~~ = recipes.recipe class ID ~~in the recipes table~~

SQL

```
SELECT RecipeTitle, Preparation,
   RecipeClassDescription
FROM Recipe_Classes INNER JOIN Recipes
  ON Recipe_Classes.RecipeClassID =
    Recipes.RecipeClassID
```

❖ **说明**　你可能注意到了，在整理结果中，我让措词更准确地指出了在 SQL 中需要的子句。创建复杂的查询时，建议你使用这种方法，这将简化从整理到 SQL 的过渡。

在 FROM 子句中使用多个表时，务必使用表名来限定列名，从而明确地指出是哪个表的哪列（现在你明白了花时间解释列引用的原因了吧）。注意，在 ON 子句中，我对列名 RecipeClassID 做了限定，因为有两列都名为 RecipeClassID，其中一列在 Recipes 表中，另一列在 Recipe_Classes 表中。对于 SELECT 子句中的 RecipeTitle、Preparation 和 RecipeClassDescription，不用进行限定，因为这些列都只出现在一个表中。如果要在输出中包含 RecipeClassID，必须告诉数据库系统是哪个 RecipeClassID——Recipe_Classes 表中的还是 Recipes 表中的。要在查询中限定所有列名，应像下面这样做：

SQL

```
SELECT Recipes.RecipeTitle,
   Recipes.Preparation,
   Recipe_Classes.RecipeClassDescription
FROM Recipe_Classes INNER JOIN Recipes
  ON Recipe_Classes.RecipeClassID =
    Recipes.RecipeClassID
```

❖ **说明**　虽然大多数商业 SQL 实现支持关键字 JOIN，但有些不支持。如果你使用的数据库系统不支持 JOIN，依然可以解决前面的问题，方法是在 FROM 子句中列出所需的所有表，并将 ON 子句中的查找条件移到 WHERE 子句中。在不支持 JOIN 的数据库系统中，可像下面这样解决这个问题：

```
SELECT Recipes.RecipeTitle, Recipes.Preparation,
   Recipe_Classes.RecipeClassDescription
FROM Recipe_Classes, Recipes
WHERE Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID
```

对初学者来说，创建简单查询时，这种语法可能直观得多。然而，SQL 标准语法允许完全在 FROM 子句中指定最终结果集的来源。可认为 FROM 子句全面地指定了一个连接结果集，数据库系统将从中获取答案。在 SQL 标准中，WHERE 子句只用来将某些行从 FROM 定义的结果集中过滤掉。

不太难，不是吗？但图 8-2 列出的 USING 子句有何用途呢？如果两个表中要匹配的列的名称相同，且要做的连接基于值相等，就可使用 USING 子句并在其中列出列名。下面使用 USING 来解决前面的问题。

“列出数据库中所有菜品的名称、制作方法和菜品类型描述。”

转换	Select recipe title, preparation, and recipe class description from the recipe classes table joined with the recipes table using recipe class ID（使用菜品类型 ID 连接菜品类型表和菜品表，并选择菜品名、制作方法和菜品类型描述）
整理	Select recipe title, preparation, ~~and~~ recipe class description from ~~the~~ recipe classes ~~table~~ inner join~~ed with the~~ recipes ~~table~~ using recipe class ID

SQL	SELECT Recipes.RecipeTitle, Recipes.Preparation, Recipe_Classes.RecipeClassDescription FROM Recipe_Classes INNER JOIN Recipes USING (RecipeClassID)

有些数据库系统不支持 USING。在你使用的数据库系统中，如果发现不能使用 USING，可使用 ON 子句和相等比较来获得相同的效果。

❖ **说明**　SQL 标准还定义了自然连接（NATURAL JOIN），它通过匹配所有同名列来连接两个表。如果你使用的数据库系统支持 NATURAL JOIN，可像下面这样解决这个示例问题：

```
SELECT Recipes.RecipeTitle, Recipes.Preparation,
   Recipe_Classes.RecipeClassDescription
FROM Recipe_Classes
NATURAL INNER JOIN Recipes
```

使用了关键字 NATURAL 时，不要指定 ON 或 USING 子句。别忘了，关键字 INNER 是可选的；如果你指定的是 NATURAL JOIN，则默认为内连接。

本节前面说过，数据库系统在逻辑上将第一个表的每行与第二个表的每行合并，再应用 ON 或 USING 中指定的条件。看起来好像数据库系统需要做大量额外的工作：先生成所有的组合，再筛选出可能为数不多的匹配行。

但请放心，所有现代关系型数据库系统都在评估完整个 JOIN 子句后才开始取回行。在前面一直使用的示例中，很多数据库系统都这样解析这个请求：先从 Recipe_Classes 表中取回一行，再使用内部链接，即索引（如果表的设计者定义了）在 Recipes 表中快速找出与之匹配的所有行，然后再接着处理 Recipe_Classes 表中的下一行。换而言之，通过利用灵巧（优化）的方案，数据库系统只取回匹配的行。数据库表只包含几百行时，这看起来不重要，但需要处理数十万行时，将有很大的不同！

2. 给表指定关联名（别名）

对于 FROM 子句中列出的表，SQL 标准定义了给它们指定别名［SQL 标准中称为**关联名**（correlation name）］的方法。创建复杂的查询时，如果使用的表的名称很长，这项功能将提供极大的便利。对于名称很长的表，可给它指定简短的关联名，这样将更容易引用其中的列。

图 8-4 演示了如何在 FROM 子句中给表指定关联名。

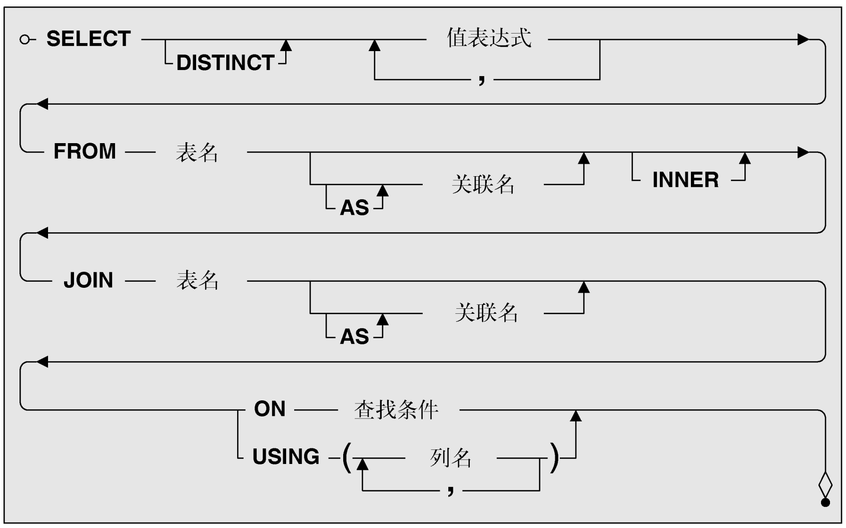


图 8-4　在 FROM 子句中给表指定关联名（别名）

要给表指定关联名，可在表名后加上可选的关键字 AS，再加上要指定的关联名（与所有可选的关键字一样，建议不要省略 AS，这样查询将更容易阅读和理解）。

给表指定关联名后，就可在其他所有子句中使用关联名来替代原来的表名，这包括 SELECT 子句、ON 和 WHERE 子句的查找条件以及 ORDER BY 子句。这可能令人困惑，因为你通常先编写 SELECT 子句，再编写 FROM 子句。如果你打算在 FROM 子句中给表指定表名，则在 SELECT 子句中限定列名时必须使用该别名。

下面来重写前面的示例查询，在其中使用关联名，让你看看结果是什么样的。这里给 Recipes 表指定别名 R，给 Recipe_Classes 表指定别名 RC，如下所示：

```
SQL        SELECT R.RecipeTitle, R.Preparation,
              RC.RecipeClassDescription
           FROM Recipe_Classes AS RC
             INNER JOIN Recipes AS R
               ON RC.RecipeClassID = R.RecipeClassID
```

假设你要添加一个筛选器，以便只列出类别为主菜（Main course）或甜品（Dessert）的菜品（有关如何定义筛选器的详情，请参阅第 6 章）。指定关联名后，引用相应的表时必须使用新指定的名称。下面是所需的 SQL：

```
SQL        SELECT R.RecipeTitle, R.Preparation,
              RC.RecipeClassDescription
           FROM Recipe_Classes AS RC
             INNER JOIN Recipes AS R
               ON RC.RecipeClassID = R.RecipeClassID
           WHERE RC.RecipeClassDescription = 'Main course'
           OR RC.RecipeClassDescription = 'Dessert'
```

并非必须给所有的表都指定关联名。在前面的示例中，可只给 Recipes 表或 Recipe_Classes 表指定关联名。

在复杂的连接中，有时必须给表指定关联名。为提供这样的示例，我们来看 Bowling League 数据库。图 8-5 说明了 Teams 表和 Bowlers 表之间的关系。

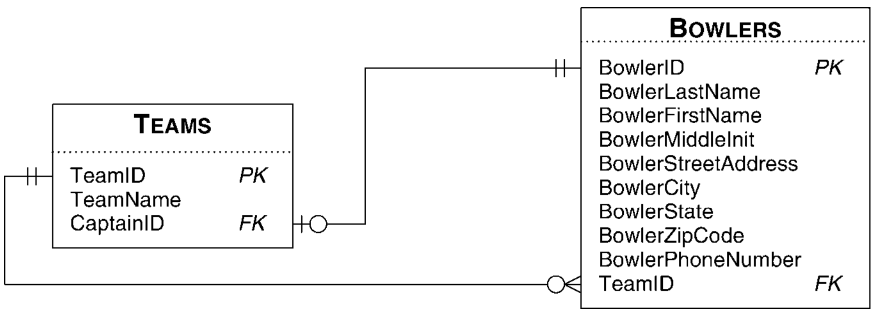


图 8-5　Teams 表和 Bowlers 表之间的关系

如你所见，Bowlers 表中的 TeamID 是一个外键，让你能够找到球队中所有投球手的信息。球队队长也是投球手，因此也可从 Bowlers 表的 BowlerID 连接到 Teams 表的 CaptainID。

如果你要使用单个查询来列出球队名、球队队长的姓名以及所有投球手的姓名，就必须在查询中两次使用 Bowlers 表，一次连接到 CaptainID 以检索球队队长的姓名，另一次连接到 TeamID 以检索所有队员。在这种情况下，必须在其中一次使用 Bowlers 表时给它指定别名，或两次使用时都指定别名，这样数据库系统才能将获取队长姓名的那次和获取所有队员的那次区分开来。本章后面将提供一个示例，它要求多次使用同一个表并指定别名；这个示例使用的是 Bowling League 数据库，可在 8.4.2 节中找到。

3. 嵌入 SELECT 语句

来看一种更有趣的情况。在大多数 SQL 实现中，可将 FROM 子句中的表名替换为完整的 SELECT 语句。在 SQL 标准中，像这样嵌入的 SELECT 语句被称为**派生表**（derived table）。只要想一想就知道，SELECT 语句不过是一种获

取一个或多个表中部分数据的方式。当然，必须指定关联名，赋予嵌入查询的结果一个名称。图 8-6 说明了如何在 JOIN 子句中嵌入 SELECT 语句。

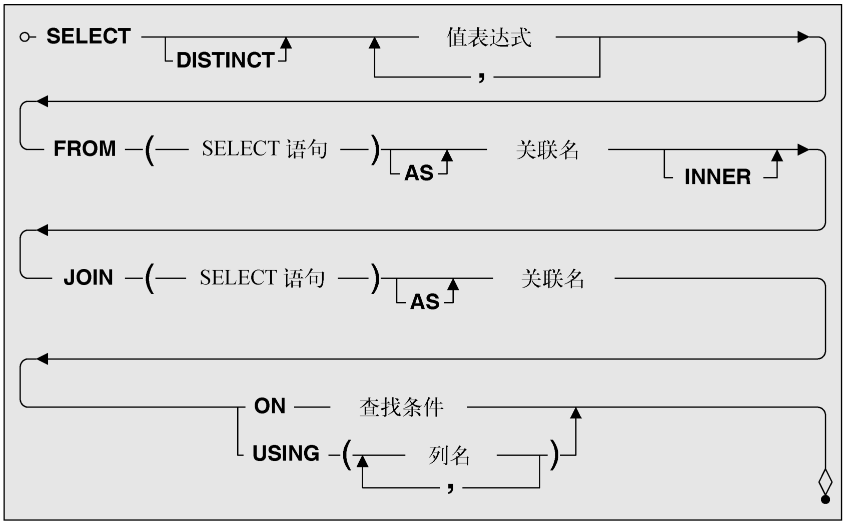


图 8-6　将 JOIN 中的表名替换为 SELECT 语句

请注意，在嵌入的 SELECT 语句中，可包含除 ORDER BY 子句外的其他所有查询子句。另外，在关键字 INNER JOIN 的两边，可混合使用 SELECT 语句和表名。

回到 Recipes 和 Recipe_Classes 表，并假设请求依然是只取回主菜和甜品。这里重写了查询，在 INNER JOIN 子句中使用一条 SELECT 语句从 Recipe_Classes 表中筛选行：

```
SQL          SELECT R.RecipeTitle, R.Preparation,
                RCFiltered.ClassName
             FROM
               (SELECT RecipeClassID,
                   RecipeClassDescription AS ClassName
                FROM Recipe_Classes AS RC
                WHERE RC.ClassName = 'Main course' OR
                   RC.ClassName = 'Dessert') AS RCFiltered
             INNER JOIN Recipes AS R
               ON RCFiltered.RecipeClassID = R.RecipeClassID
```

❖ **说明**　有些数据库系统不支持在 FROM 子句中嵌入 SELECT 语句。如果你使用的数据库系统是这样的，可将嵌入的 SELECT 语句保存为视图，再使用这个视图的名称替换嵌入的 SELECT 语句。

请注意，决定用 SELECT 语句替换表名时，务必包含要加入到最终结果中的列以及执行连接操作所需的列。这就是前述嵌入的 SELECT 语句中包含 RecipeClassID 和 RecipeClassDescription 的原因所在。出于好玩，我在嵌入的 SELECT 语句中给 RecipeClassDescription 指定了别名，因此外面的 SELECT 语句要求取回 ClassName 而不是 RecipeClassDescription。注意，在 ON 子句中，引用的是嵌入的 SELECT 语句的关联名 RCFiltered，而不是原来的表名，也不是在嵌入的 SELECT 语句中给这个表指定的关联名。

如果你使用的数据库系统配备了非常聪明的优化器，那么这样定义查询时，执行速度将与前面的做法（在 JOIN 后面使用 WHERE 子句根据 RecipeClassDescription 进行筛选）相同。为了以最高效的方式处理请求，数据库系统应先筛选 Recipe_Classes 表中的行，再在 Recipes 表中查找匹配的行。如果先将 Recipe_Classes 表中的每行与 Recipes 表中匹配的行连接起来，再进行筛选，速度将慢得多。如果你发现处理请求的时间不该那么长，请将 WHERE 子句移到嵌入在 JOIN 中的 SELECT 语句中，让数据库系统先筛选 Recipe_Classes 表中的行。

4. 在 JOIN 中嵌入 JOIN

虽然通过连接两个表可解决很多问题，但为获得所需的数据，经常需要连接三个、四个乃至更多的表。例如，你可能想使用单个查询取回关于菜品的所有相关信息：菜品的类型、名称和所有食材。图 8-7 显示了为处理这个请求需要用到哪些表。

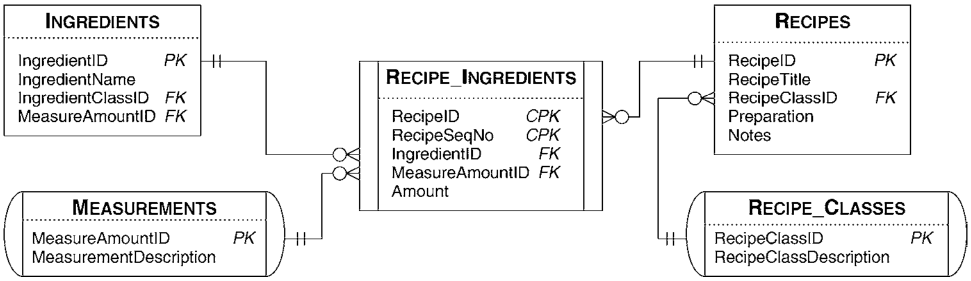


图 8-7　为取回所有有关菜品的信息，需要用到示例数据库 Recipes 中的哪些表

看起来要从 5 个表中获取信息（其中的 Measurements 和 Recipe_Classes 是查找表，因此在图中以不同的方式表示）。别怕，要完成这项任务，可编写更复杂的 FROM 子句——在 JOIN 子句中嵌入 JOIN 子句。诀窍如下：在每个可指定表名的地方，都可指定一个用括号括起的 JOIN 子句。图 8-8 对图 8-4 进行了简化（省略了关联名子句并使用 ON 子句，以呈现简单的两表连接）。

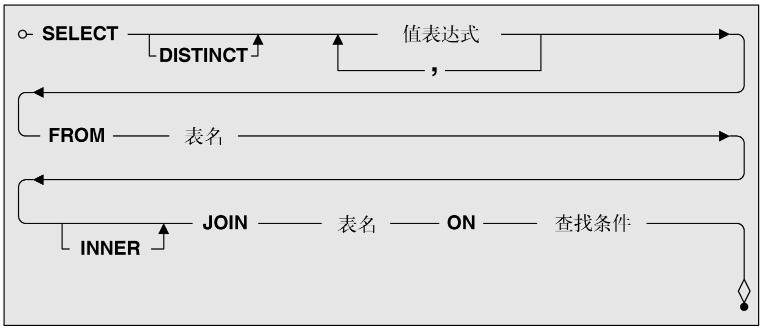


图 8-8　简单的两表内连接

要添加第三个表，只需在第一个表名前添加左括号，再在查找条件后面添加右括号、INNER JOIN、表名、关键字 ON 和查找条件，如图 8-9 所示。

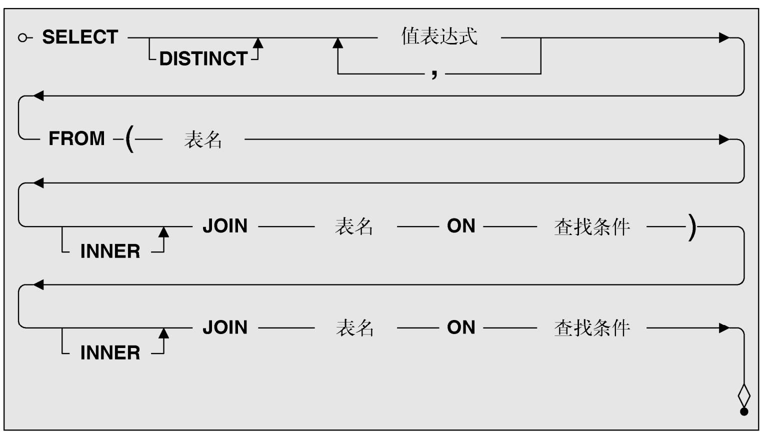


图 8-9　简单的三表内连接

只要想一想就知道，括号内的两表内连接是一个逻辑表，即内部结果集。这个结果集取代了图 8-8 中的第一个表名。你可以不断重复这个过程（将整个 JOIN 子句用括号括起，再添加关键字 JOIN、表名、关键字 ON 和查找条件），直到获得了所有需要的结果集。下面来看一个请求，它要求从图 8-7 所示的所有表中获取数据。

“我需要从数据库 Recipes 中获取菜品类型、菜品名、制作说明、食材名、食材序号、食材数量和食材度量单位，并按菜品名和序号排序。”

转换	Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table joined with the recipes table on recipe class ID in the recipe classes table matching recipe class ID in the recipes table, then joined with the recipe ingredients table on recipe ID in the recipes table matching recipe ID in the recipe ingredients table, then joined with the ingredients table on ingredient ID in the ingredients table matching ingredient ID in the recipe ingredients table, and then finally joined with the measurements table on measurement amount ID in the measurements table matching measurement amount ID in the recipe ingredients table, order by recipe title and recipe sequence number（依次基于菜品类型 ID 相等将菜品类型表连接到菜品表、基于菜品 ID 相等连接到菜品食材表、基于食材 ID 相等连接到食材表、基于度量单位 ID 连接到度量单位表，选择菜品类型描述、菜品名、制作说明、食材名、食材序号、食材数量和食材度量单位，并按菜品名和序号排序）
整理	Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description from ~~the~~ recipe classes ~~table~~ inner join~~ed with the~~ recipes ~~table~~ on recipe_classes.recipe class ID ~~in the recipe classes table matching~~ = recipes.recipe class ID ~~in the recipes table, then~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matching~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table, then~~ inner join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matching~~ = ingredients.ingredient ID ~~in the recipe ingredients table, and then finally~~ inner join~~ed with the~~ measurements ~~table~~ on measurements.measurement amount ID ~~in the measurements table matching~~ = recipe ingredients.measurement amount ID ~~in the recipe ingredients table,~~ order by recipe title and recipe sequence number

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle, Recipes.Preparation,
    Ingredients.IngredientName,
    Recipe_Ingredients.RecipeSeqNo,
    Recipe_Ingredients.Amount,
    Measurements.MeasurementDescription
FROM (((Recipe_Classes
  INNER JOIN Recipes
  ON Recipe_Classes.RecipeClassID =
     Recipes.RecipeClassID)
  INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
  INNER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID)
  INNER JOIN Measurements
  ON Measurements.MeasureAmountID =
     Recipe_Ingredients.MeasureAmountID
ORDER BY RecipeTitle, RecipeSeqNo
```

如果还要筛选出主菜，该怎么办呢？如果你说为此需要在 ORDER BY 前面添加一个 WHERE 子句，那么你猜对了。

实际上，在可放置表名的任何地方，都可用一个两表连接取而代之。图 8-9 隐含着这样的意思，即必须先将前两个表连接，再将结果与第 3 个表连接，但完全可以先将后两个表连接（只要第 3 个表与第 2 个表而不是第 1 个表相关），再与第 1 个表连接。图 8-10 说明了这种方法。

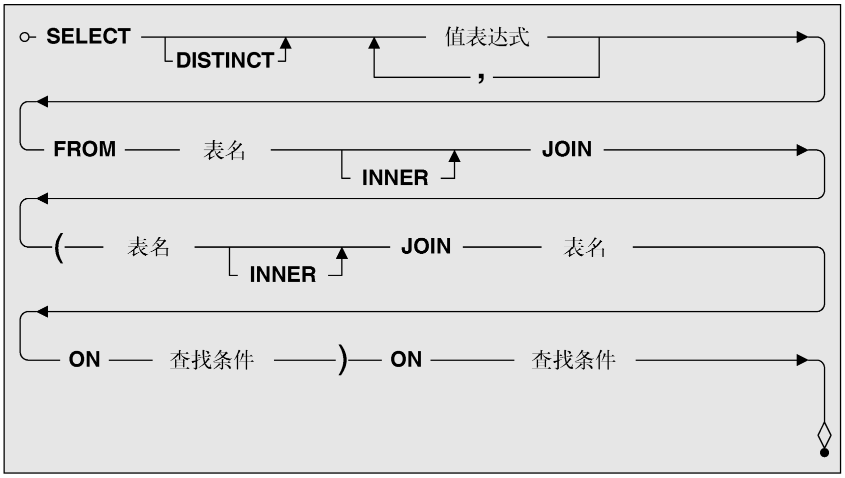


图 8-10　以另一种顺序连接两个以上的表

我们从绘画的角度来看看这个问题。如果要混合出崧蓝绿色，混合顺序并不那么重要。要得到这种颜色，可先混合白色和蓝色得到浅蓝色，再混入黄色，也可先混合蓝色和黄色得到绿色，再混入白色。

为使用 5 个表处理刚说的请求，也可编写下面这样的 SQL：

```
SQL         SELECT Recipe_Classes.RecipeClassDescription,
               Recipes.RecipeTitle, Recipes.Preparation,
               Ingredients.IngredientName,
               Recipe_Ingredients.RecipeSeqNo,
               Recipe_Ingredients.Amount,
               Measurements.MeasurementDescription
            FROM Recipe_Classes
              INNER JOIN (((Recipes
              INNER JOIN Recipe_Ingredients
                ON Recipes.RecipeID =
                   Recipe_Ingredients.RecipeID)
              INNER JOIN Ingredients
              ON Ingredients.IngredientID =
                 Recipe_Ingredients.IngredientID)
              INNER JOIN Measurements
              ON Measurements.MeasureAmountID =
                 Recipe_Ingredients.MeasureAmountID)
              ON Recipe_Classes.RecipeClassID =
                 Recipes.RecipeClassID
            ORDER BY RecipeTitle, RecipeSeqNo
```

你必须明白这一点，因为在他人编写的查询或 Query By Example 软件编写的 SQL 中，你可能遇到这样的构造。另外，有些数据库系统的优化器对连接定义的顺序很敏感，如果你发现使用了大量连接的查询在大型数据库中的执行时间很长，可调整 SQL 语句中连接的顺序，这样或许能够缩短查询的执行时间。为简单起见，本章后面的大部分示例都将沿简单路径进行连接，即基于附录 B 的示例数据库模式图按从左到右、从上到下的顺序连接各个表。

8.2.4　检查表间关系

知道表之间的关系至关重要，这一点现在应该显而易见。如果你发现所需的数据列在不同的表中，可能需要编写像刚才演示的 FROM 子句那样复杂的 FROM 子句，这样才能以合乎逻辑的方式获取所有的信息。在这种情况下，如果不知道表之间的关系以及用于建立关系的列，你就会陷入困境。

在很多情况下，可能需要穿越多个关系才能获得想要的数据。例如，下面来简化前面的请求，只获取菜品名和食材名。

“列出所有菜品的名称以及制作每种菜品所需的食材。”

<table>
<tr><td>转换</td><td>Select the recipe title and the ingredient name from the recipes table joined with the recipe ingredients table on recipe ID in the recipes table matching recipe ID in the recipe ingredients table, and then joined with the ingredients table on ingredient ID in the ingredients table matching ingredient ID in the recipe ingredients table（基于菜品 ID 相等将菜品表连接到菜品食材表，再基于食材 ID 相等连接到食材表，并选择菜品名和食材名）</td></tr>
<tr><td>整理</td><td>Select ~~the~~ recipe title ~~and the~~ ingredient name from ~~the~~ recipes ~~table~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matching~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table, and then~~ inner join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matching~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~</td></tr>
<tr><td>SQL</td><td><pre>
SELECT Recipes.RecipeTitle,
   Ingredients.IngredientName
FROM (Recipes
INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
   Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
  ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID
</pre></td></tr>
</table>

注意，虽然你不需要 Recipe_Ingredients 表的任何列，但依然必须在查询中包含这个表。这是因为要将 Recipes 和 Ingredients 表关联起来，必须通过 Recipe_Ingredients 表。

8.3　内连接的用途

对内连接的机制有了基本的认识后，来看看使用它可解决的一些问题。

8.3.1　查找相关的行

你知道，内连接最常见的用途是连接表，让你能够从不同的相关表中取回列。下面以示例数据库为例，列出了使用内连接可处理的请求类型。

“列出供应商及其供应的商品。”
“列出员工及其为哪些顾客下了订单。”
“列出经纪人及其签订的演出合约的演出日期。”
“列出顾客及其签约的演唱组合。”
“查找为顾客 Berg 或 Hallmark 演出过的演唱组合。”
“显示大楼及其所有教室。”
“列出教员及其讲授的科目。”
“列出保龄球队及其队长的姓名。”
“列出保龄球队及其所有队员。”
“列出包含牛肉或大蒜的菜品。”
“显示包含胡萝卜的菜品使用的所有食材。”

8.4 节将演示如何编写查询来处理类似这样的请求（以及其他请求）。

8.3.2　查找匹配的值

内连接还有一个不那么常见的用途，就是在多个表或结果集中找出这样的行，即它们的一个或多个非相关键列的值匹配。我在第 7 章承诺过，后面将介绍如何使用内连接替代 INTERSECT。下面是使用这种方法可处理的一些请求。

“列出姓名相同的顾客和员工。”
“列出居住在同一座城市的顾客和员工。“

“找出所有订购了自行车和头盔的顾客。”
“找出居住在同一个邮区的经纪人和演唱组合。”
“列出为顾客 Bonnicksen 和 Rosales 都演出过的演唱组合。”
“列出名字相同的学生和老师。”
“列出艺术和计算机科学课程的平均成绩都不低于 85 的学生。”
“找出居住在同一个邮区（邮政编码相同）的投球手。”
“找出在保龄球馆 Thunder bird Lanes 和 Bolero Lanes 比赛时原始得分都不低于 155 的投球手。”
“找出度量单位相同的食材。”
“列出包含牛肉和大蒜的菜品。”

下一节将演示如何解决几个类似于这样的问题。

8.4　语句举例

知道如何使用内连接来编写查询以及使用内连接可处理的请求类型后，来看一些使用内连接的示例。这些示例都来自示例数据库，演示了如何使用内连接从两个或更多表中取回数据，以及如何通过匹配值来解决问题。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。

每个查询都存储在相应的示例数据库中，查询名以 CH08 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明**　这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。

8.4.1　两个表

我将从简单的开始，先列举只需连接两个表的请求。

- **Sales Orders 数据库**

“显示所有商品及其所属的类别。”

转换/整理：Select category description ~~and~~ product name from ~~the~~ categories ~~table~~ inner join~~ed with the~~ products ~~table~~ on categories.category ID ~~in the categories table matching~~ = products.category ID ~~in the products table~~（基于类别 ID 相等将类别表内连接到商品表，并选择类别描述和商品名）

SQL：
```
SELECT Categories.CategoryDescription,
   Products.ProductName
FROM Categories
  INNER JOIN Products
    ON Categories.CategoryID =
       Products.CategoryID
```

CH08_Products_And_Categories（40 行）

CategoryDescription	ProductName
Accessories	Dog Ear Cyclecomputer
Accessories	Dog Ear Helmet Mount Mirrors
Accessories	Viscount C-500 Wireless Bike Computer
Accessories	Kryptonite Advanced 2000 U-Lock
Accessories	Nikoma Lok-Tight U-Lock

（续）

CategoryDescription	ProductName
Accessories	Viscount Microshell Helmet
Accessories	Viscount CardioSport Sport Watch
Accessories	Viscount Tru-Beat Heart Transmitter
Accessories	Dog Ear Monster Grip Gloves
	<< 其他行 >>

❖ **说明** 别忘了，由于没有包含 ORDER BY 子句，在不同数据库系统返回的结果集中，行的排列顺序可能不同。你在上面看到的是 Microsoft Access 和 MySQL 中的排列顺序，因为这两个数据库系统好像都先从 Categories 表中取回行，再在 Products 表中查找匹配的行，因此行是按类别名排列的。在 Microsoft SQL Server 和 PostgreSQL 中，先从 Products 表中取回行，再在 Categories 表中查找匹配的行，因此排列顺序不同。如果没有使用 ORDER BY 子句，不同数据库系统返回行的顺序就可能不同。

- **Entertainment Agency 数据库**

“列出演唱组合及其演出合约的开始日期、结束日期和价格。”

转换/整理	Select entertainer stage name, start date, end date, ~~and~~ contract price from ~~the~~ entertainers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on entertainers.entertainer ID ~~in the entertainers table matching~~ = engagements.entertainer ID ~~in the engagements table~~（基于演唱组合 ID 相等将演唱组合表内连接到演出合约表，并选择艺名、开始日期、结束日期和合约价格）

SQL

```
SELECT Entertainers.EntStageName,
   Engagements.StartDate, Engagements.EndDate,
   Engagements.ContractPrice
FROM Entertainers
  INNER JOIN Engagements
    ON Entertainers.EntertainerID =
       Engagements.EntertainerID
```

CH08_Entertainers_And_Contracts（111 行）

EntStageName	StartDate	EndDate	ContractPrice
Carol Peacock Trio	2017-09-18	2017-09-26	$1,670.00
Carol Peacock Trio	2017-10-01	2017-10-07	$1,940.00
Carol Peacock Trio	2017-10-14	2017-10-15	$410.00
Carol Peacock Trio	2017-10-21	2017-10-21	$140.00
Carol Peacock Trio	2017-11-13	2017-11-19	$680.00
Carol Peacock Trio	2017-12-23	2017-12-26	$410.00
Carol Peacock Trio	2017-12-29	2018-01-07	$1,400.00
Carol Peacock Trio	2018-01-08	2018-01-08	$320.00
Carol Peacock Trio	2018-01-22	2018-01-30	$1,670.00
Carol Peacock Trio	2018-02-11	2018-02-19	$1,670.00
Carol Peacock Trio	2018-02-25	2018-02-28	$770.00
Topazz	2017-09-11	2017-09-18	$770.00
	<< 其他行 >>		

- **School Scheduling 数据库**

“列出周三上课的课程所属的科目。”

转换/整理	Select subject name from ~~the~~ subjects table inner join~~ed with the~~ classes ~~table~~ on subjects.subject ID ~~in the subjects table matching~~ = classes.subject ID ~~in the classes table~~ where Wednesday schedule ~~is~~ = true（基于科目 ID 相等将科目表内连接到课程表，并选择周三上课的科目名）

SQL

```
SELECT DISTINCT Subjects.SubjectName
FROM Subjects
INNER JOIN Classes
  ON Subjects.SubjectID
   = Classes.SubjectID
WHERE Classes.WednesdaySchedule = -1
```

❖ **说明**　由于同一个课程的不同部分可能安排在同一天上课，因此我使用了关键字 DISTINCT 来消除重复行。有些数据库支持关键字 TRUE，但我选择使用更通用的“所有二进制位都为 1 的整数值——–1。如果你使用的数据库系统使用一个二进制位存储真/假值，也可使用 1 来检查是否为真值。假值总是为数字零（0）。

CH08_Subjects_On_Wednesday（34 行）

SubjectName
Advanced English Grammar
Art History
Biological Principles
Chemistry
Composition—Fundamentals
Composition—Intermediate
Design
Drawing
Elementary Algebra
<< 其他行 >>

- **Bowling League 数据库**

“列出保龄球队及其队长的姓名。”

转换/整理　Select team name ~~and~~ captain full name from ~~the~~ teams ~~table~~ inner join~~ed with the~~ bowlers ~~table~~ on team captain ID ~~equals~~ = bowler ID（基于队长 ID 与投球手 ID 相等将球队表连接到投球手表，并选择球队名和队长姓名）

SQL

```
SELECT Teams.TeamName, (Bowlers.BowlerLastName
   || ', ' || Bowlers.BowlerFirstName)
      AS CaptainName
FROM Teams
INNER JOIN Bowlers
  ON Teams.CaptainID = Bowlers.BowlerID
```

CH08_Teams_And_Captains（10 行）

TeamName	CaptainName
Marlins	Fournier, David
Sharks	Patterson, Ann
Terrapins	Viescas, Carol
Barracudas	Sheskey, Richard
Dolphins	Viescas, Suzanne
Orcas	Thompson, Sarah
Manatees	Viescas, Michael
Swordfish	Rosales, Joe
Huckleberrys	Viescas, David
MintJuleps	Hallmark, Alaina

- **Recipes 数据库**

“列出包含牛肉或大蒜的菜品。”

转换/整理	Select ~~unique~~ distinct recipe title from ~~the~~ recipes ~~table~~ join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matching~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table~~ where ingredient ID ~~is~~ in ~~the list of beef and garlic IDs~~ (1, 9)（基于菜品 ID 相等将菜品表内连接到菜品食材表，并选择食材 ID 与牛肉或大蒜的食材 ID 相等的不同菜品名）

SQL

```
SELECT DISTINCT Recipes.RecipeTitle
FROM Recipes
INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID
WHERE Recipe_Ingredients.IngredientID IN (1, 9)
```

❖ **说明**　由于有些菜品包含牛肉和大蒜，为了消除可能出现的重复行，我使用了关键字 DISTINCT。

CH08_Beef_Or_Garlic_Recipes（5 行）

RecipeTitle
Asparagus
Garlic Green Beans
Irish Stew
Pollo Picoso
Roast Beef

8.4.2　超过两个表

下面来点更刺激的，发出一些要求连接超过两个表的请求。

- **Sales Orders 数据库**

“找出所有订购了自行车头盔的顾客。”

转换/整理	Select customer first name, customer last name from ~~the~~ customers ~~table~~ inner join~~ed with the~~ orders ~~table~~ on customers.customer ID ~~in the customers table matching~~ = orders.customer ID ~~in the orders table, then~~ inner join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table matching~~ = order_details.order number ~~in the order details table, then~~ inner join~~ed with the~~ products ~~table~~ on products.product number ~~in the products table matching~~ = order_details.product number ~~in the order details table~~ where product name ~~contains~~ LIKE ‘%Helmet%’（依次基于顾客 ID 相等将顾客表内连接到订单表、基于订单号相等内连接到订单详情表、基于商品号内连接到商品表，并选择商品名包含 Helmet 的顾客的名和姓）

SQL

```
SELECT DISTINCT Customers.CustFirstName,
   Customers.CustLastName
FROM ((Customers INNER JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
  ON Orders.OrderNumber =
     Order_Details.OrderNumber)
INNER JOIN Products
  ON Products.ProductNumber =
     Order_Details.ProductNumber
WHERE Products.ProductName LIKE '%Helmet%'
```

❖ **警告**　如果你使用的数据库系统在字符列中查找时区分大小写，指定查找条件时必须小心，确保使用正确的大小写。例如，在很多数据库系统中，helmet 和 Helmet 是不同的。

❖ **说明**　由于顾客可能多次订购头盔，因此我使用了关键字 DISTINCT 来消除重复行。

CH08_Customers_Who_Ordered_Helmets（25 行）

CustFirstName	CustLastName
Andrew	Cencini
Angel	Kennedy
Caleb	Viescas
Darren	Gehring
David	Smith
Dean	McCrae
Estella	Pundt
Gary	Hallmark
Jim	Wilson
John	Viescas
<< 其他行 >>	

- **Entertainment Agency 数据库**

“找出为顾客 Berg 或 Hallmark 演出过的演唱组合。”

转换/整理

Select ~~unique~~ distinct entertainer stage name from ~~the~~ entertainers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on entertainers.entertainer ID ~~in the entertainers table matching~~ = engagements.entertainer ID ~~in the engagements table, then~~ inner join~~ed with the~~ customers ~~table~~ on customers.customer ID ~~in the customers table matching~~ = engagements.customer ID ~~in the engagements table~~ where ~~the~~ customer last name ~~is~~ = ‘Berg’ or ~~the~~ customer last name ~~is~~ = ‘Hallmark’（依次基于演唱组合 ID 相等将演唱组合表内连接到演出合约表、基于顾客 ID 相等内连接到顾客表，并从顾客姓为 Berg 或 Hallmark 的行中选择不同艺名）

SQL

```
SELECT DISTINCT Entertainers.EntStageName
FROM (Entertainers
INNER JOIN Engagements
   ON Entertainers.EntertainerID =
      Engagements.EntertainerID)
INNER JOIN Customers
   ON Customers.CustomerID =
      Engagements.CustomerID
WHERE Customers.CustLastName = 'Berg'
   OR Customers.CustLastName = 'Hallmark'
```

CH08_Entertainers_For_Berg_OR_Hallmark（8 行）

EntStageName
Carol Peacock Trio
Coldwater Cattle Company
Country Feeling
Jim Glynn
JV & the Deep Six
Modern Dance
Susan McLain
Topazz

- **Bowling League 数据库**

“列出所有的联赛、场次和各局的结果。”

转换/整理

Select tourney ID, tourney location, match ID, lanes, odd lane team, even lane team, game number, game winner from ~~the~~ tournaments ~~table~~ inner join~~ed with the~~ tourney matches ~~table~~ on tournaments.tourney ID ~~in the tournaments table matching~~ = tourney_matches.tourney ID ~~in the tourney matches table, then~~ inner join~~ed with the~~ teams ~~table aliased~~ as odd team on

oddteam.team ID ~~in the odd team table matches~~ = tourney_matches.odd lane team ID ~~in the tourney matches table, then~~ inner join~~ed with the~~ teams ~~table aliased~~ as even team on eventeam.team ID ~~in the even team table matches~~ = tourney_matches.even lane team ID ~~in the tourney matches table, then~~ inner join~~ed with the~~ match games table on match_games.match ID ~~in the match games table matches~~ = tourney_matches.match ID ~~in the tourney matches table, then~~ inner join~~ed with the~~ teams ~~table aliased~~ as winner on winner.team ID ~~in the winner table matches~~ = match_games.winning team ID ~~in the match games table~~［依次基于联赛 ID 相等将联赛表内连接到场次表、基于单数赛道球队 ID 和球队 ID 相等连接到球队表（别名为单数赛道球队表）、基于双数赛道球队 ID 和球队 ID 相等内连接到球队表（别名为双数赛道球队表）、基于场次 ID 相等内连接到局次表、基于球队 ID 相等内连接到球队表（别名为获胜球队表），并选择联赛 ID、联赛地点、场次 ID、赛道、单数赛道球队、双数赛道球队、局次和获胜球队］

SQL

```sql
SELECT Tournaments.TourneyID AS Tourney,
  Tournaments.TourneyLocation AS Location,
  Tourney_Matches.MatchID,
  Tourney_Matches.Lanes,
  OddTeam.TeamName AS OddLaneTeam,
  EvenTeam.TeamName AS EvenLaneTeam,
  Match_Games.GameNumber AS GameNo,
  Winner.TeamName AS Winner
FROM (((Tournaments
  INNER JOIN Tourney_Matches
    ON Tournaments.TourneyID
    = Tourney_Matches.TourneyID)
  INNER JOIN Teams AS OddTeam
    ON OddTeam.TeamID
    = Tourney_Matches.OddLaneTeamID)
  INNER JOIN Teams AS EvenTeam
    ON EvenTeam.TeamID
    = Tourney_Matches.EvenLaneTeamID)
  INNER JOIN Match_Games
    ON Match_Games.MatchID
    = Tourney_Matches.MatchID)
  INNER JOIN Teams AS Winner
    ON Winner.TeamID
    = Match_Games.WinningTeamID
```

❖ **说明**　这个查询很有意思，它需要使用同一个表（Teams）三次。为确保查询合法，必须至少在两次使用这个表时指定关联名，但我在三次使用时都指定了别名，以清楚地指出每次使用时，这个表在查询中扮演着什么角色。

CH08_Tournament_Match_Game_Results（168 行）

Tourney	Location	MatchID	Lanes	OddLane Team	EvenLane Team	GameNo	Winner
1	Red Rooster Lanes	1	01-02	Marlins	Sharks	1	Marlins
1	Red Rooster Lanes	1	01-02	Marlins	Sharks	2	Sharks
1	Red Rooster Lanes	1	01-02	Marlins	Sharks	3	Marlins
1	Red Rooster Lanes	2	03-04	Terrapins	Barracudas	1	Terrapins
1	Red Rooster Lanes	2	03-04	Terrapins	Barracudas	2	Barracudas
1	Red Rooster Lanes	2	03-04	Terrapins	Barracudas	3	Terrapins
1	Red Rooster Lanes	3	05-06	Dolphins	Orcas	1	Dolphins

（续）

Tourney	Location	MatchID	Lanes	OddLane Team	EvenLane Team	GameNo	Winner
1	Red Rooster Lanes	3	05-06	Dolphins	Orcas	2	Orcas
1	Red Rooster Lanes	3	05-06	Dolphins	Orcas	3	Dolphins
			<< 其他行 >>				

❖ **说明**　虽然这里的记录是按联赛和场次排列的，但这只是我使用的数据库系统（Microsoft Access）返回记录的顺序。如果要确保记录按特定的顺序排列，必须使用 ORDER BY 子句。

- **Recipes 数据库**

“列出主菜及其使用的所有食材。”

转换/整理	Select recipe title, ingredient name, measurement description, ~~and~~ amount from ~~the~~ recipe classes ~~table~~ inner join~~ed with the~~ recipes ~~table~~ on recipes.recipe class ID ~~in the recipes table matches~~ = recipe_classes.recipe class ID ~~in the recipe classes table~~, then inner join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matches~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table, then~~ inner join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table, and finally~~ inner join~~ed with the~~ measurements ~~table~~ on measurements.measure amount ID ~~in the measurements table matches~~ = recipe_ingredients.measure amount ID ~~in the recipe ingredients table~~, where recipe class description is = ‘Main course’（依次基于菜品类型 ID 相等将菜品类型表内连接到菜品表、基于菜品 ID 相等内连接到菜品食材表、基于食材 ID 相等内连接到食材表、基于度量单位 ID 相等内连接到度量单位表，并选择菜品类型描述为 Main course 的菜品名、食材名、度量单位描述和数量）

SQL

```
SELECT Recipes.RecipeTitle,
   Ingredients.IngredientName,
   Measurements.MeasurementDescription,
   Recipe_Ingredients.Amount
FROM (((Recipe_Classes
INNER JOIN Recipes
  ON Recipes.RecipeClassID =
     Recipe_Classes.RecipeClassID)
INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID)
INNER JOIN Measurements
  ON Measurements.MeasureAmountID =
     Recipe_Ingredients.MeasureAmountID
WHERE Recipe_Classes.RecipeClassDescription
      = 'Main course'
```

❖ **警告**　Ingredients 和 Recipe_Ingredients 表都包含 MeasureAmountID 列。如果定义最后的连接时，基于 Ingredients 表而不是 Recipe_Ingredients 表中的 MeasureAmountID 列，得到的将是食材的默认度量单位，而不是菜品中给食材指定的度量单位。

CH08_Main_Course_Ingredients（53 行）

RecipeTitle	IngredientName	Measurement Description	Amount
Irish Stew	Beef	Pound	1
Irish Stew	Onion	Whole	2
Irish Stew	Potato	Whole	4
Irish Stew	Carrot	Whole	6

（续）

RecipeTitle	IngredientName	Measurement Description	Amount
Irish Stew	Water	Quarts	4
Irish Stew	Guinness Beer	Ounce	12
Fettuccini Alfredo	Fettuccini Pasta	Ounce	16
Fettuccini Alfredo	Vegetable Oil	Tablespoon	1
Fettuccini Alfredo	Salt	Teaspoon	3
	<< 其他行 >>		

8.4.3　查找匹配的值

最后，我们让图片呈现立体状态。在刚才的一系列请求中，都是基于相同的列来连接多个结果集或表（如果你使用的数据库系统支持关键字 INTERSECT，上述很多问题也都可通过计算结果集的交集来解决）。

- **Sales Orders 数据库**

“找出所有订购了自行车和头盔的顾客。”

这个请求看起来足够简单——也许太简单了。我们换种方式提问，更清楚地指出需要让数据库系统做什么。

“找出所有订购了自行车的顾客，再找出所有订购了头盔的顾客，然后列出这两个结果集中都有的顾客，他们就是订购了自行车和头盔的顾客。”

第 1 次转换	Select customer first name and customer last name from those common to the set of customers who ordered bicycles and the set of customers who ordered helmets（选择同时出现在订购了自行车的顾客集和订购了头盔的顾客集中的顾客的名和姓）
第 2 次转换/整理	Select customer first name ~~and~~ customer last name from (Select ~~unique~~ distinct customer name, customer first name, customer last name from ~~the~~ customers ~~table~~ inner join~~ed with the~~ orders ~~table~~ on customers.customer ID ~~in the customers table matches~~ = orders.customer ID ~~in the orders table, then~~ inner join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table matches~~ = order_details.order number ~~in the order details table~~, then inner join~~ed with the~~ products table on products.product number ~~in the products table matches~~ = order_details.product number ~~in the order details table~~ where product name ~~contains~~ LIKE ‘%Bike’) as cust bikes inner join~~ed with~~ (Select ~~unique~~ distinct customer ID from ~~the~~ customers ~~table~~ inner join~~ed with the~~ orders ~~table~~ on customers.customer ID ~~in the customers table matches~~ = orders.customer ID in ~~the~~ orders ~~table,~~ then inner join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table matches~~ = order_details.order number ~~in the order details table, then~~ join~~ed with the~~ products table on products.product number in ~~the products table matches~~ = order_details.product number ~~in the order details table~~ where product name ~~contains~~ LIKE ‘%Helmet’) as cust helmets on cust bikes.customer ID ~~in the cust bikes table matches~~ = cust helmets.customer ID ~~in the cust helmets table~~（依次基于顾客 ID 相同将顾客表内连接到订单表、基于订单号相同内连接到订单详情表、基于商品号相同内连接到商品表，再选择商品名包含 Bike 的商品对应的不同顾客的 ID、名和姓，并将该结果集命名为 CustBikes；依次基于顾客 ID 相同将顾客表内连接到订单表、基于订单号相同内连接到订单详情表、基于商品号相同内连接到商品表，再选择商品名包含 Helmet 的商品对应的不同顾客的 ID，并将该结果集命名为 CustHelmets；基于顾客 ID 相同将这两个结果集内连接，并选择顾客的名和姓）

SQL

```
SELECT CustBikes.CustFirstName,
   CustBikes.CustLastName
FROM
   (SELECT DISTINCT Customers.CustomerID,
      Customers.CustFirstName,
      Customers.CustLastName
    FROM ((Customers
    INNER JOIN Orders
      ON Customers.CustomerID
        = Orders.CustomerID)
    INNER JOIN Order_Details
      ON Orders.OrderNumber =
        Order_Details.OrderNumber)
    INNER JOIN Products
      ON Products.ProductNumber =
      Order_Details.ProductNumber
    WHERE Products.ProductName LIKE '%Bike')
  AS CustBikes
```

```
            INNER JOIN
              (SELECT DISTINCT Customers.CustomerID
                FROM ((Customers
                INNER JOIN Orders
                  ON Customers.CustomerID =
                     Orders.CustomerID)
                INNER JOIN Order_Details
                  ON Orders.OrderNumber =
                     Order_Details.OrderNumber)
                INNER JOIN Products
                  ON Products.ProductNumber =
                     Order_Details.ProductNumber
                WHERE Products.ProductName LIKE '%Helmet')
                  AS CustHelmets
            ON CustBikes.CustomerID =
               CustHelmets.CustomerID
```

❖ **说明**　这里简化了第二条嵌入的 SELECT 语句，使其只取回 CustomerID，因为只需使用这列就可将两个结果集内连接起来。实际上，可不内连接到 Customers 表，而从 Orders 表中获取 CustomerID。别忘了，对于嵌入在 FROM 子句中的 SELECT 语句，可将其视为一个逻辑表；对于每条这样的语句，我都给它指定了独特的名称，以便能够编写最后一条 ON 子句。

也可通过对两个集合执行交集运算来解决这个问题，但如果这样做，输出中将包含两个结果集中都有的列。坦率地说，就这个问题而言，这里演示的解决方案可能不是最佳的；第 11 章介绍如何使用子查询时，将演示如何更高效地解决这个问题。

CH08_Customers_Both_Bikes_And_Helmets（21 行）

CustFirstName	CustLastName
William	Thompson
Robert	Brown
Dean	McCrae
John	Viescas
Mariya	Sergienko
Neil	Patterson
Andrew	Cencini
Angel	Kennedy
Liz	Keyser
Rachel	Patterson
<< 其他行 >>	

- **Entertainment Agency 数据库**

“列出为 Berg 和 Hallmark 都演出过的演唱组合。”

你在前面看到过，要列出为 Berg 或 Hallmark 演出过的演唱组合很容易。下面来修改这个请求的措词，更清楚地指出需要让数据库系统做什么。

“找出为 Berg 演出过的所有演唱组合，再找出为 Hallmark 演出过的所有演唱组合，然后列出在这两个结果集中都出现了的演唱组合，他们就是为 Berg 和 Hallmark 都演出过的演唱组合。”

第 1 次转换	Select entertainer stage name from those common to the set of entertainers who played for Berg and the set of entertainers who played for Hallmark（选择同时出现在为 Berg 演出过的演唱组合集和为 Hallmark 演出过的演唱组合集中的演唱组合的艺名）
第 2 次转换/整理	Select entertainer stage name from (Select ~~unique~~ distinct entertainer stage name from ~~the~~ entertainers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on entertainers.entertainer ID ~~in the entertainers table matches~~ = engagements.entertainer ID ~~in the engagements table, then~~ inner join~~ed with the~~ customers ~~table~~ on customers.customer ID ~~in the customers table matches~~ = engagements.customer

ID ~~in the engagements table~~ where customer last name ~~is~~ = 'Berg') as entberg inner join~~ed with~~ (Select ~~unique~~ distinct entertainer stage names from ~~the~~ entertainers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on entertainers.entertainer ID ~~in the entertainers table matches~~ = engagements.entertainer ID ~~in the engagements table, then~~ join~~ed with the~~ customers ~~table~~ on customers.customer ID in the customers ~~table matches~~ = engagements.customer ID ~~in the engagements table~~ where customer last name ~~is~~ = 'Hallmark') as enthallmark on entberg.entertainer ID ~~in the entberg table matches~~ = enthallmark.entertainer ID ~~in the enthallmark table~~（依次基于演唱组合 ID 相同将演唱组合表内连接到演出合约表、基于顾客 ID 相同内连接到顾客表，再选择姓 Berg 的顾客对应的不同演唱组合的 ID 和艺名，并将该结果集命名为 EntBerg；依次基于演唱组合 ID 相同将演唱组合表内连接到演出合约表、基于顾客 ID 相同内连接到顾客表，再选择姓 Hallmark 的顾客对应的不同演唱组合的 ID 和艺名，并将该结果集命名为 EntHallmark；基于演唱组合 ID 相同内连接这两个结果集，并选择艺名）

SQL

```
SELECT EntBerg.EntStageName
FROM
   (SELECT DISTINCT Entertainers.EntertainerID,
      Entertainers.EntStageName
    FROM (Entertainers
    INNER JOIN Engagements
      ON Entertainers.EntertainerID =
         Engagements.EntertainerID)
    INNER JOIN Customers
      ON Customers.CustomerID =
         Engagements.CustomerID
    WHERE Customers.CustLastName = 'Berg')
AS EntBerg INNER JOIN
  (SELECT DISTINCT Entertainers.EntertainerID,
      Entertainers.EntStageName
   FROM (Entertainers
   INNER JOIN Engagements
     ON Entertainers.EntertainerID =
        Engagements.EntertainerID)
   INNER JOIN Customers
     ON Customers.CustomerID =
        Engagements.CustomerID
   WHERE Customers.CustLastName = 'Hallmark')
  AS EntHallmark
ON EntBerg.EntertainerID =
    EntHallmark.EntertainerID
```

CH08_Entertainers_Berg_AND_Hallmark（4 行）

EntStageName
Carol Peacock Trio
JV & the Deep Six
Modern Dance
Country Feeling

❖ **说明** 这个请求也可使用 INTERSECT 来解决；使用第 11 章将介绍的子查询也可更高效地解决它。

- **School Scheduling 数据库**

"列出名字相同的学生和老师。"

转换/整理

Select student full name ~~and~~ staff full name from ~~the~~ students ~~table~~ inner join~~ed with the~~ staff ~~table~~ on students.first name ~~in the students table matches~~ = staff.first name ~~in the staff table~~（基于名字相同将学生表内连接到教员表，并选择学生的姓名和教员的姓名）

SQL

```
SELECT (Students.StudFirstName || ' ' ||
   Students.StudLastName) AS StudFullName,
   (Staff.StfFirstName || ' ' ||
   Staff.StfLastName) AS StfFullName
FROM Students
INNER JOIN Staff
  ON Students.StudFirstName = Staff.StfFirstName
```

CH08_Students_Staff_Same_FirstName（2 行）

StudFullName	StfFullName
Michael Viescas	Michael Hernandez
David Hamilton	David Smith

- **Bowling League 数据库**

“找出在保龄球馆 Thunderbird Lanes 和 Bolero Lanes 比赛时，原始得分都不低于 170 的投球手。”

这也是一个可使用内连接来解决的交集问题。下面换种提问方式，更清楚地指出需要让数据库系统做什么。

“找出在保龄球馆 Thunderbird Lanes 比赛时，原始得分不低于 170 的所有投球手；再找出在保龄球馆 Bolero Lanes 比赛时，原始得分不低于 170 的所有投球手；然后列出同时出现在这两个结果集中投球手，他们就是在这两个保龄球馆比赛时成绩都不错的投球手。”

第 1 次转换

Select bowler full name from those common to the set of bowlers who have a score of 170 or better at Thunderbird Lanes and the set of bowlers who have a score of 170 or better at Bolero Lanes（选择在保龄球馆 Thunderbird Lanes 比赛时原始得分不低于 170 的投球手集以及在保龄球馆 Bolero Lanes 比赛时原始得分不低于 170 的投球手集中都出现了的投球手的姓名）

第 2 次转换/整理

Select bowler full name from (Select ~~unique~~ distinct bowler ID ~~and~~ bowler full name from ~~the~~ bowlers ~~table~~ inner join~~ed with the~~ bowler scores ~~table~~ on bowlers.bowler ID ~~in the bowlers table matches~~ = bowler_scores.bowler ID ~~in the bowler scores table, then~~ inner join~~ed with the~~ tourney matches table on tourney_matches.match ID ~~in the tourney matches table matches~~ = bowler_scores.match ID ~~in the bowler scores table, and finally~~ inner join~~ed with the~~ tournaments ~~table~~ on tournaments.tourney ID ~~in the tournaments table matches~~ = tourney_matches.tourney ID ~~in the tourney matches table~~ where tourney location ~~is~~ = 'Thunderbird Lanes' and raw score ~~is greater than or equal to~~ >= 170) as bowlertbird inner join~~ed with~~ (Select ~~unique~~ distinct bowler ID ~~and~~ bowler full name from ~~the~~ bowlers ~~table~~ inner join~~ed with the~~ bowler scores ~~table~~ on bowlers.bowler ID ~~in the bowlers table matches~~ = bowler_scores.bowler ID in the bowler scores tab~~le, then~~ inner join~~ed with the~~ tourney matches table on tourney_matches.match ID ~~in the tourney matches table matches~~ = bowler_scores.match ID ~~in the bowler scores table, and finally~~ inner join~~ed with the tournaments table~~ on tournaments.tourney ID ~~in the tournaments table matches~~ = tourney_matches.tourney ID in the tourney matches table where tourney location is = 'Bolero Lanes' and raw score ~~is greater than or equal to~~ >= 170) as bowlerbolero on bowlertbird.bowler ID ~~in the bowlertbird table matches~~ = bowlerbolero.bowler ID ~~in the bowlerbolero table~~（依次基于投球手 ID 相同将投球手得分表内连接到投球手表、基于场次 ID 相同内连接到场次表、基于联赛 ID 内连接到联赛表，再选择联赛地点为 Thunderbird Lanes 且原始得分不低于 170 的不同投球手的 ID 和姓名，并将该结果集命名为 BowlerTbird；依次基于投球手 ID 相同将投球手得分表内连接到投球手表、基于场次 ID 相同内连接到场次表、基于联赛 ID 内连接到联赛表，再选择联赛地点为 Bolero Lanes 且原始得分不低于 170 的不同投球手的 ID 和姓名，并将该结果集命名为 BowlerBolero；基于投球手 ID 相同将这两个结果集内连接，并投球手姓名）

SQL

```
SELECT BowlerTbird.BowlerFullName
FROM
  (SELECT DISTINCT Bowlers.BowlerID,
     (Bowlers.BowlerLastName || ', ' ||
     Bowlers.BowlerFirstName) AS BowlerFullName
   FROM ((Bowlers
   INNER JOIN Bowler_Scores
   ON Bowlers.BowlerID = Bowler_Scores.
BowlerID)
   INNER JOIN Tourney_Matches
    ON Tourney_Matches.MatchID =
       Bowler_Scores.MatchID)
   INNER JOIN Tournaments
    ON Tournaments.TourneyID =
       Tourney_Matches.TourneyID
   WHERE Tournaments.TourneyLocation =
       'Thunderbird Lanes'
    AND Bowler_Scores.RawScore >= 170)
  AS BowlerTbird INNER JOIN
    (SELECT DISTINCT Bowlers.BowlerID,
       (Bowlers.BowlerLastName || ', ' ||
       Bowlers.BowlerFirstName) AS
BowlerFullName
    FROM ((Bowlers
    INNER JOIN Bowler_Scores
    ON Bowlers.BowlerID = Bowler_Scores.
```

```
BowlerID)
    INNER JOIN Tourney_Matches
    ON Tourney_Matches.MatchID =
       Bowler_Scores MatchID)
    INNER JOIN Tournaments
    ON Tournaments.TourneyID =
       Tourney_Matches.TourneyID
    WHERE Tournaments.TourneyLocation =
       'Bolero Lanes'
   AND Bowler_Scores.RawScore >= 170)
  AS BowlerBolero
ON BowlerTbird.BowlerID = BowlerBolero.BowlerID
```

❖ **说明** 由于同一名投球手在这两个保龄球馆比赛时可能多次得分不低于 170，为消除重复的行，我添加了关键字 DISTINCT。同样，对于这个问题，更佳的解决方案可能是使用第 11 章将学习的子查询。

CH08_Good_Bowlers_TBird_And_Bolero（11 行）

BowlerFullName
Kennedy, John
Patterson, Neil
Kennedy, Angel
Patterson, Kathryn
Viescas, John
Viescas, Caleb
Thompson, Sarah
Thompson, Mary
Thompson, William
Patterson, Rachel
Clothier, Ben

- **Recipes 数据库**

"显示包含胡萝卜的菜品的所有食材。"

转换/整理

Select recipe ID, recipe title, ~~and~~ ingredient name from ~~the~~ recipes ~~table~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matches~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table~~, inner join~~ed with the ingredients table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table, then finally~~ inner join~~ed with~~ (Select recipe ID from ~~the~~ ingredients ~~table~~ inner join~~ed with the recipe ingredients table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~ where ingredient name ~~is~~ = 'Carrot') as carrots on recipes.recipe ID ~~in the recipes table matches~~ = carrots.recipe ID ~~in the carrots table~~（依次基于菜品 ID 相同将菜品表内连接到菜品食材表、基于食材 ID 相同内连接到食材表；基于食材 ID 相同将食材表内连接到菜品食材表，再选择名为 Carrot 的食材所属菜品的 ID，并将该结果集命名为 Carrots；基于菜品 ID 相同内连接这两个结果集，并选择菜品 ID、菜品名和食材名）

SQL

```
SELECT Recipes.RecipeID, Recipes.RecipeTitle,
   Ingredients.IngredientName
FROM ((Recipes
INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
    Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
  ON Ingredients.IngredientID =
    Recipe_Ingredients.IngredientID)
INNER JOIN
   (SELECT Recipe_Ingredients.RecipeID
    FROM Ingredients
    INNER JOIN Recipe_Ingredients
      ON Ingredients.IngredientID =
         Recipe_Ingredients.IngredientID
    WHERE Ingredients.IngredientName = 'Carrot')
  AS Carrots
ON Recipes.RecipeID = Carrots.RecipeID
```

❖ **说明**　这个请求也可使用子查询来解决，具体方法将在第 11 章演示。

CH08_Recipes_Containing_Carrots（16 行）

RecipeID	RecipeTitle	IngredientName
1	Irish Stew	Beef
1	Irish Stew	Onion
1	Irish Stew	Potato
1	Irish Stew	Carrot
1	Irish Stew	Water
1	Irish Stew	Guinness Beer
14	Salmon Filets in Parchment Paper	Salmon
14	Salmon Filets in Parchment Paper	Carrot
14	Salmon Filets in Parchment Paper	Leek
14	Salmon Filets in Parchment Paper	Red Bell Pepper
14	Salmon Filets in Parchment Paper	Butter
	<< 其他行 >>	

8.5　小结

本章详细讨论了如何基于匹配的值来连接多个表或结果集。首先定义了连接的概念，然后详细讨论了内连接。我讨论了可基于哪些内容来连接，同时警告你不要建立不合乎逻辑的连接。

我首先列举了一些连接两个表的示例，然后演示了如何给 FROM 子句中的表指定关联名（别名）。有时指定关联名只是为了方便，而有时必须指定关联名，如多次使用了同一个表或使用了嵌入的 SELECT 语句时。

我演示了如何在 FROM 子句中将表引用替换为 SELECT 语句，然后演示了如何连接两个以上的表或结果集，以拓展你的视野；最后讨论了内连接的语法，并再次重申了如下两点的重要性：数据库必须设计良好；必须明白数据库中各表之间的关系。

我讨论了一些让内连接很有用的原因，并列举了具体的示例。在本章余下的篇幅中，我列举了十多个内连接使用示例，它们被分成三类：连接两个表的；连接两个以上表的；基于匹配值的连接。下一章将探讨另一种连接——外连接。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

8.6　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 列出顾客及其下单日期，并按下单日期排序。
  提示：需要连接两个表。
  解决方案见 CH08_Customers_And_OrderDates（944 行）。

  (2) 列出员工及其为哪些顾客下了订单。
  提示：需要连接两个以上的表。
  解决方案见 CH08_Employees_And_Customers（211 行）。

  (3) 显示所有的订单、每个订单包含的商品以及每种商品的库存量，并按订单号排序。
  提示：需要连接两个以上的表。
  解决方案见 CH08_Orders_With_Products（3973 行）。

(4) 显示供应商及其提供的价格低于 100 美元的商品。
提示：需要连接两个以上的表。
解决方案见 CH08_Vendors_And_Products_Less_Than_100（66 行）。
(5) 列出同姓的顾客和员工。
提示：需要基于匹配值建立连接。
解决方案见 CH08_Customers_Employees_Same_LastName（16 行）。
(6) 列出居住在同一座城市的顾客和员工。
提示：需要基于匹配值建立连接。
解决方案见 CH08_Customers_Employees_Same_City（10 行）。

- **Entertainment Agency 数据库**

(1) 显示经纪人及其签订的演出合约的日期，并按演出合约的起始日期排序。
提示：需要连接两个表。
解决方案见 CH08_Agents_Booked_Dates（111 行）。
(2) 列出顾客及其签约过的演唱组合。
提示：需要连接两个以上的表。
解决方案见 CH08_Customers_Booked_Entertainers（75 行）。
(3) 找出居住地邮政编码相同的经纪人和演唱组合。
提示：需要基于匹配值建立连接。
解决方案见 CH08_Agents_Entertainers_Same_Postal（10 行）。

- **School Scheduling 数据库**

(1) 显示所有的教学楼及其中的教室。
提示：需要连接两个表。
解决方案见 CH08_Buildings_Classrooms（47 行）。
(2) 列出学生及其当前注册的课程。
提示：需要连接两个以上的表。
解决方案见 CH08_Student_Enrollments（50 行）。
(3) 列出教工及其讲授的科目。
提示：需要连接两个以上的表。
解决方案见 CH08_Staff_Subjects（110 行）。
(4) 列出艺术和计算机课程的成绩都不低于 85 分的学生。
提示：需要基于匹配值建立连接。
解决方案见 CH08_Good_Art_CS_Students（1 行）。

- **Bowling League 数据库**

(1) 列出所有保龄球队及其队员。
提示：需要连接两个表。
解决方案见 CH08_Teams_And_Bowlers（32 行）。
(2) 显示投球手及其参加的比赛场次和得分。
提示：需要连接两个以上的表。
解决方案见 CH08_Bowler_Game_Scores（1344 行）。
(3) 找出居住地邮政编码相同的投球手。
提示：需要基于匹配值建立连接，同时务必避免投球手与自己匹配。
解决方案见 CH08_Bowlers_Same_ZipCode（92 行）。

- **Recipes 数据库**

  (1) 列出所有属于沙拉的菜品。
  提示：需要连接两个表。
  解决方案见 CH08_Salads（1 行）。

  (2) 列出所有包含奶制品的菜品。
  提示：需要连接两个以上的表。
  解决方案见 CH08_Recipes_Containing_Dairy（2 行）。

  (3) 找出默认度量单位相同的食材。
  提示：需要基于匹配值建立连接。
  解决方案见 CH08_Ingredients_Same_Measure（628 行）。

  (4) 显示包含牛肉和大蒜的菜品。
  提示：需要基于匹配值建立连接。
  解决方案见 CH08_Beef_And_Garlic_Recipes（1 行）。

第 9 章

外 连 接

“问题和解决方案的唯一差别在于后者是人们能够看懂的。”

——发明家查尔斯·凯特灵（1876—1958）

本章涵盖如下主题：

- 何谓外连接
- 左/右外连接
- 全外连接
- 外连接的用途
- 语句举例
- 小结
- 练习

前一章全面介绍了内连接，即使用 INNER JOIN 将多个表或结果集关联起来，找出所有匹配的行。本章介绍外连接，即通过将表关联起来，找出匹配的行和不匹配的行。

9.1 何谓外连接

前一章说过，SQL 标准定义了多种将多个表或结果集关联起来的连接操作。外连接（OUTER JOIN）让数据库系统根据指定的准则返回匹配的行，同时返回被关联的两个表或一个表中不匹配的行。

例如，假设你要从数据库 School Scheduling 获取有关学生及其注册的课程的信息。前一章介绍过，内连接只返回注册了课程的学生以及有学生注册了的课程，而不会返回还没有注册任何课程的学生，也不会返回没有任何学生注册的课程。

如果你要列出所有的学生及其注册的课程（如果有的话），该怎么办呢？或者相反，如果你要列出所有的课程以及注册各个课程的学生（如果有的话），又该怎么办呢？要解决这类问题，需要使用外连接。

图 9-1 使用集合图说明了学生与课程之间的一种可能关系。从中可知，有些学生没有注册任何课程，而有些课程没有任何学生注册。

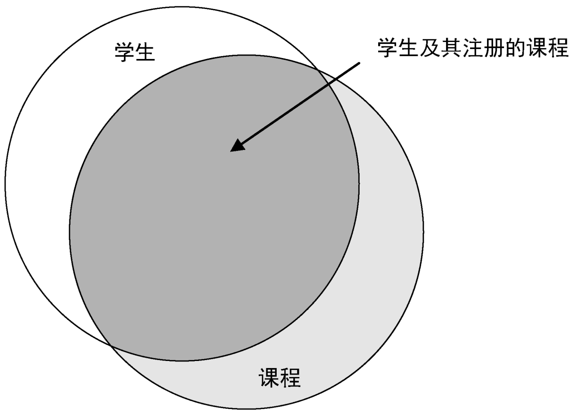


图 9-1 学生与课程之间的关系可能是这样的

如果你要求获取所有学生及其注册的课程，结果集将如图 9-2 所示。

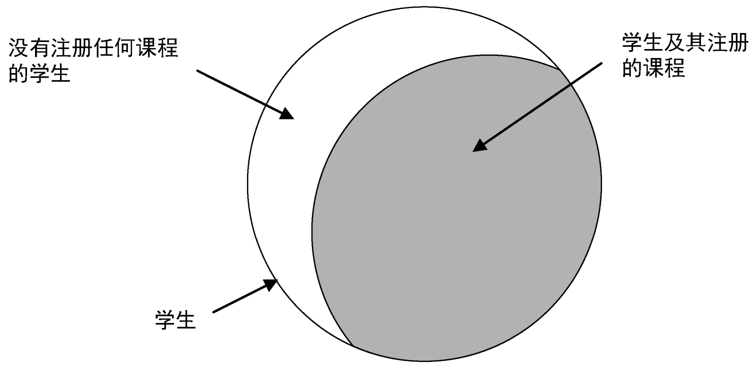


图 9-2 所有的学生及其注册的课程

你可能会问：对于没有注册任何课程的学生，结果集中对应的行是什么样的呢？如果你还记得第 5 章介绍的 Null（空值），就知道结果集是什么样的：如果你要求获取所有学生及其注册的课程，则对于还没有注册任何课程的学生，数据库系统将把返回的 Classes 表的所有列都设置为 Null。只要想一想第 7 章讨论的差集概念，你就知道没有注册任何课程的学生其实就是如下两个集合的差集：由所有学生组成的集合以及由至少注册了一门课程的学生组成的集合。

同理，如果你要求获取所有的课程以及注册各门课程的学生，则在返回的结果集中，将有一些这样的行，即其包含的 Students 表中的列的值为 Null。这些行是如下两个集合的差集：由所有课程组成的集合以及由有学生注册了的课程组成的集合。本书前面说过，通过结合使用外连接和 Null 值测试，也可计算出两个集合的差集。真正的 EXCEPT 操作找出两个集合中完全匹配的行，而连接操作与此不同，它只要求特定的列（通常是主键和外键）匹配。

9.2 左/右外连接

通常使用外连接来获取一个表或结果集中的所有行以及另一个表或结果集中匹配的行，为此，可使用左外连接（LEFT OUTER JOIN）或右外连接（RIGHT OUTER JOIN）。

这两种外连接有何不同呢？前一章介绍过，内连接两个表时，依次指定第一个表、关键字 JOIN 和第二个表。当你使用外连接来创建查询时，SQL 标准认为第一个表在左边，第二个表在右边。因此，如果你要返回第一个表的所有行以及第二个表中匹配的行，就使用左外连接；相反，如果你要返回第二个表的所有行以及第一个表中匹配的行，就使用右外连接。

语法

下面来看看左/右外连接的语法。

1. 使用表

先从简单的开始，定义一个使用表的外连接。图 9-3 所示的语法图演示了如何通过外连接两个表来创建查询。

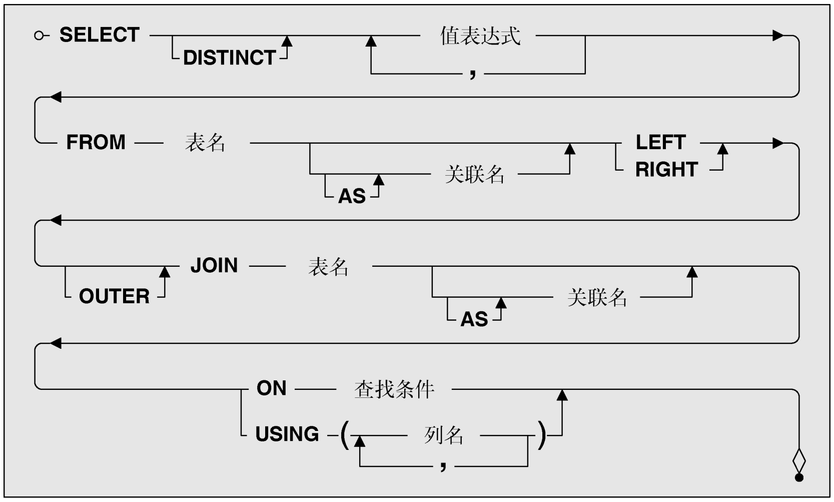


图 9-3　外连接两个表

与第 8 章介绍的内连接一样，所有操作都是在 FROM 子句中进行的（出于简化考虑，这里暂时省略了 WHERE 和 ORDER BY 子句）。创建连接两个表的查询时，你指定两个表，并使用关键字 JOIN 将它们关联起来。如果没有指定连接的类型，数据库系统将认为是内连接。在这里，你要执行的是外连接，因此必须显式地指定 LEFT JOIN 或 RIGHT JOIN。关键字 OUTER 是可选的。

❖ **说明**　如果你查看附录 A 中完整的语法图，将发现我将相关的部分（SELECT 语句、表引用和连接表）合而为一了。

在任何连接中，查找条件部分都是一个 ON 或 USING 子句，它跟在第二个表后面，告诉数据库系统如何执行连接操作。从逻辑上说，为执行连接操作，数据库系统将合并两个表的各行，再根据 ON 或 USING 子句中的条件找出并返回匹配的行。由于你要求执行的是外连接，因此数据库系统也将返回左表或右表中不匹配的行。

第 6 章介绍了如何使用查找条件来编写 WHERE 子句。在连接操作的 ON 子句中，可使用查找条件来指定逻辑测试，这样仅当它为真时才返回关联的行。编写查找条件时，必须至少对第一个表的一列和第二个表的一列进行比较，这样才合乎逻辑。虽然可编写非常复杂的查找条件，但通常使用简单的相等比较测试，检查一个表的主键与另一个表的外键是否相等。

为简单起见，我们先来看前一章使用的菜品类型和菜品示例。本书前面说过，为确保数据库设计良好，必须将复杂的类型名放在一个独立的表中，并通过简单的键值将类型名关联到主表。在示例数据库 Recipes 中，菜品类型和菜品存储在不同的表中，图 9-4 显示了 Recipe_Classes 表和 Recipes 表之间的关系。

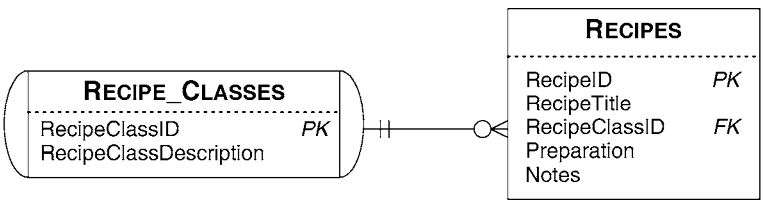


图 9-4　菜品类型和菜品存储在不同的表中

最初创建这个数据库时，你可能首先输入所有能想到的菜品类型。等输入大量菜品后，你可能想知道哪些类型没有任何菜品，还可能想列出所有的菜品类型以及每种类型包含的菜品。这两个问题都可使用外连接来解决。

❖ **说明**　贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”方法。

“列出所有的菜品类型以及各类型包含的菜品。”

转换	Select recipe class description and recipe title from the recipe classes table left outer joined with the recipes table on recipe class ID in the recipe classes table matching recipe class ID in the recipes table（基于类型 ID 相同将菜品类型表左外连接到菜品表，并选择菜品类型描述和菜品名）
整理	Select recipe class description ~~and~~ recipe title from ~~the~~ recipe classes ~~table~~ left outer join~~ed with the~~ recipes ~~table~~ on recipe_classes.recipe class ID ~~in the recipe classes table matching~~ = recipes.recipe class ID ~~in the recipes table~~
SQL	SELECT Recipe_Classes.RecipeClassDescription, Recipes.RecipeTitle FROM Recipe_Classes LEFT OUTER JOIN Recipes ON Recipe_Classes.RecipeClassID = Recipes.RecipeClassID

在 FROM 子句中使用多个表时，别忘了使用表名限定每个列名，清楚地指出是哪个表中的列。注意，在 ON 子句中，我对列名 RecipeClassID 做了限定，因为有两个名为 RecipeClassID 的列，一个位于 Recipes 表中，另一个位于 Recipe_Classes 表中。

❖ **说明**　虽然大多数 SQL 商业实现支持 OUTER JOIN，但有一些不支持。如果你使用的数据库系统不支持 OUTER JOIN，可这样解决这个问题：在 FROM 子句中列出所需的所有表，再将查找条件从 ON 子句移到 WHERE 子句。有关在你使用的数据库系统中定义外连接的非标准语法，请参阅其文档。例如，较早的 Microsoft SQL Server 版本支持下面这样的语法（请注意 WHERE 子句中的星号）：

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle
FROM Recipe_Classes, Recipes
WHERE Recipe_Classes.RecipeClassID *=
   Recipes.RecipeClassID
```

如果你使用的是 Oracle，可选语法如下（请注意 WHERE 子句中的加号）：

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle
FROM Recipe_Classes, Recipes
WHERE Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID(+)
```

坦率地说，这些怪异语法是数据库厂商发明的，旨在提供外连接功能，因为那时在 SQL 标准定义中还没有定义更清晰的语法。多亏了 SQL 标准语法，可在 FROM 子句中全面定义最终结果集的来源。你可认为 FROM 子句完整地定义了一个连接结果集，而数据库系统将从中获取你所需的答案。在 SQL 标准中，WHERE 子句只用于从 FROM 子句定义的结果集中排除行。另外，使用 WHERE 子句来定义外连接的语法因产品而异，因此如果你需要使用多种非标准产品，可能就需要学习多种语法。

如果你在示例数据库 Recipes 中执行这个查询，将返回 16 行。由于我还没有在这个数据库中输入任何汤类菜品，因此在 RecipeClassDescription 为 Soup 的行中，RecipeTitle 列的值为 Null。下面介绍如何找出这一行。

"列出不包含任何菜品的菜品类型。"

转换	Select recipe class description from the recipe classes table left outer joined with the recipes table on recipe class ID where recipe ID is empty（基于菜品类型 ID 相同将菜品类型表左外连接到菜品表，并基于选择菜品 ID 为空选择菜品类型描述）
整理	Select recipe class description from ~~the~~ recipe classes ~~table~~ left outer join~~ed with the~~ recipes ~~table~~ on recipe_classes.recipe class ID ~~in the recipes table matches~~ = recipes.recipe class ID ~~in the recipes table~~ where recipe ID is ~~empty~~ NULL

SQL	`SELECT Recipe_Classes.RecipeClassDescription`
	`FROM Recipe_Classes`
	`LEFT OUTER JOIN Recipes`
	`  ON Recipe_Classes.RecipeClassID =`
	`     Recipes.RecipeClassID`
	`WHERE Recipes.RecipeID IS NULL`

只要想一想就知道，这里使用连接执行了差集运算或 EXCEPT 操作（参见第 7 章），大致相当于说：列出所有的菜品类型，但出现在菜品表中的除外。图 9-5 所示的集合图有助于你理解这一点。

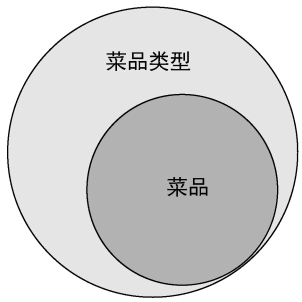


图 9-5 菜品类型和菜品之间的关系可能是这样的

在图 9-5 中，每个菜品都属于特定菜品类型，但有些菜品类型不包含任何菜品。前面的添加 IS NULL 测试相当于请求所有这样的行：位于外面的浅色圆圈内，但不与里面的深色圆圈表示的菜品集合中的任何行匹配。

注意，在图 9-3 所示的外连接语法图中，还包含可选的 USING 子句。如果两个表中要匹配的列同名，而你只想基于它们的值相等进行连接，就可使用 USING 子句并在其中列出这些列的名称。下面使用 USING 子句来解决前面的问题。

"显示不包含任何菜品的菜品类型。"

转换	Select recipe class description from the recipe classes table left outer joined with the recipes table using recipe class ID where recipe ID is empty（基于菜品类型 ID 相同将菜品类型表左外连接到菜品表，并基于选择菜品 ID 为空选择菜品类型描述）
整理	Select recipe class description from ~~the~~ recipe classes ~~table~~ left outer join~~ed with the~~ recipes ~~table~~ using recipe class ID where recipe ID is ~~empty~~ NULL

SQL	`SELECT Recipe_Classes.RecipeClassDescription`
	`FROM Recipe_Classes`
	`LEFT OUTER JOIN Recipes`
	`USING (RecipeClassID)`
	`WHERE Recipes.RecipeID IS NULL`

USING 子句的语法简单得多，不是吗？但有一点需要注意：在 USING 子句中，指定的列不属于任何表，因为 SQL 标准规定，数据库系统必须将两列合并成一列。在这个示例中，只有一个 RecipeClassID 列，因此不能在 SELECT 或其他子句中引用 Recipes.RecipeClassID 或 Recipe_Classes.RecipeClassID。

请注意，有些数据库系统不支持 USING。如果你使用的数据库系统是这样的，可使用 ON 子句和相等比较来获得同样的结果。

❖ **说明** SQL 标准还定义了一种名为自然连接（NATURAL JOIN）的连接操作，它通过匹配所有同名列来将两个表连接起来。如果你使用的数据库系统支持 NATURAL JOIN，可像下面这样来解决这个问题：

```
SELECT Recipe_Classes.RecipeClassDescription
FROM Recipe_Classes
NATURAL LEFT OUTER JOIN Recipes
WHERE Recipes.RecipeID IS NULL
```

使用了关键字 NATURAL 时，不要指定 ON 或 USING 子句。

2. 嵌入 SELECT 语句

第 8 章说过，在大多数 SQL 实现中，可将 FROM 子句中的表名替换为完整的 SELECT 语句。当然，这样做时，必须指定关联名（参见 8.2.3 节中的“给表指定关联名（别名）”部分），给嵌入的查询的结果指定一个名称。图 9-6 演示了如何使用 SELECT 语句来编写 OUTER JOIN 子句。

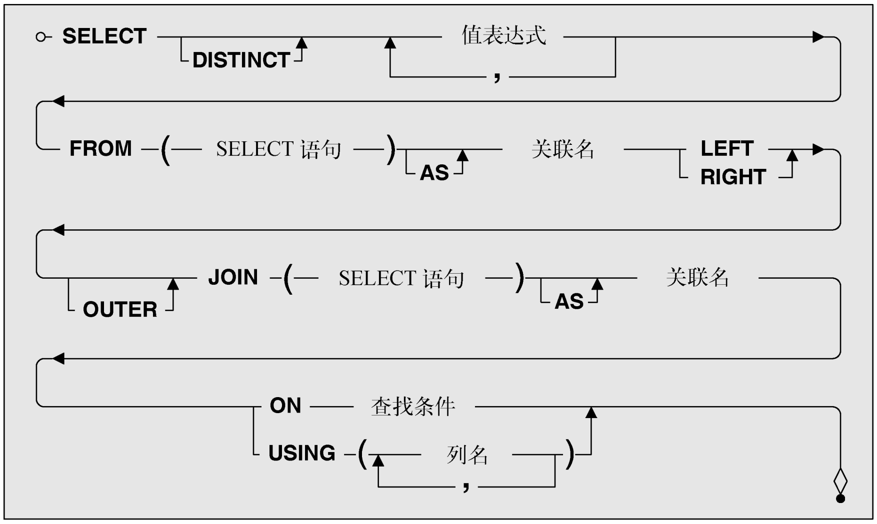


图 9-6　使用 SELECT 语句的外连接

请注意，在嵌入的 SELECT 语句中，可包含除 ORDER BY 子句外的其他所有查询子句。另外，可在关键字 OUTER JOIN 的一边使用 SELECT 语句，而在另一边使用表名。

再来看看 Recipes 和 Recipe_Classes 表。在这个示例中，假定你只对菜品类型 Salads（沙拉）、Soups（汤类）和 Main courses（主菜）感兴趣。下面的查询使用一条 SELECT 语句来筛选 Recipe_Classes 表，并将筛选结果左外连接到 Recipes 表：

```
SQL          SELECT RCFiltered.ClassName, R.RecipeTitle
             FROM
                 (SELECT RecipeClassID,
                     RecipeClassDescription AS ClassName
                 FROM Recipe_Classes AS RC
                 WHERE RC.ClassName = 'Salads'
                     OR RC.ClassName = 'Soup'
                     OR RC.ClassName = 'Main Course')
                 AS RCFiltered
             LEFT OUTER JOIN Recipes AS R
               ON RCFiltered.RecipeClassID = R.RecipeClassID
```

在 FROM 子句中使用 SELECT 语句时务必小心。首先，决定将表名替换为 SELECT 语句时，不仅要包含要出现在结果集中的列，还要包含执行连接操作所需的列。这就是前述嵌入的 SELECT 语句中包含 RecipeClassID 和 RecipeClassDescription 的原因所在。出于好玩，我在嵌入的 SELECT 语句中给 RecipeClassDescription 指定别名 ClassName，因此在 SELECT 子句中，请求的是 ClassName 而不是 RecipeClassDescription。请注意，在 ON 子句中，引用的是嵌入的 SELECT 语句的关联名（RCFiltered），而不是原表的名称，也不是我在嵌入的 SELECT 语句中给原表

指定的关联名。

如果在示例数据库中执行这个查询，将返回一行，其中的 RecipeClassDescription 和 RecipeTitle 列的值分别为 Soup 和 Null，这是因为这个示例数据库中没有属于汤类的菜品。也可在 OUTER JOIN 的右边编写一条针对 Recipes 表的 SELECT 语句。例如，可请求名称中包含单词 beef 的菜品，如下所示：

```
SQL         SELECT RCFiltered.ClassName, R.RecipeTitle
            FROM
                (SELECT RecipeClassID,
                    RecipeClassDescription AS ClassName
                FROM Recipe_Classes AS RC
                WHERE RC.ClassName = 'Salads'
                    OR RC.ClassName = 'Soup'
                    OR RC.ClassName = 'Main Course') AS RCFiltered
            LEFT OUTER JOIN
                (SELECT Recipes.RecipeClassID, Recipes.Recipe
                    Title
                FROM Recipes
                WHERE Recipes.RecipeTitle LIKE '%beef%')
              AS R
            ON RCFiltered.RecipeClassID = R.RecipeClassID
```

别忘了，左外连接获取左边的结果集或表中所有的行，而不管右边的表中是否有匹配的行。前面的查询不仅返回一个表示汤类的行（其 RecipeTitle 列的值为 Null，因为数据库中根本没有属于汤类的菜品），还返回一个表示沙拉的行（其 RecipeTitle 列的值也为 Null）。你可能由此得出结论——数据库中没有属于沙拉的菜品，但实际上，数据库中有属于沙拉的菜品，只是没有名称中包含 beef 且属于沙拉的菜品。

❖ **说明** 你可能注意到了，可在连接的 ON 子句中指定完整的查找条件。确实是这样的，因此在 SQL 标准中，完全可以像下面这样解决这个问题：

```
SELECT Recipe_Classes.RecipeClassDescription,
   Recipes.RecipeTitle
FROM Recipe_Classes
LEFT OUTER JOIN Recipes
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID
AND
   (Recipe_Classes.RecipeClassDescription = 'Salads'
OR Recipe_Classes.RecipeClassDescription = 'Soup'
OR Recipe_Classes.RecipeClassDescription =
   'Main Course')
AND Recipes.RecipeTitle LIKE '%beef%'
```

然而，我发现在有些主要的 SQL 实现中，这种做法不能正确地解决问题，甚至这种语法都是不合法的。因此，我推荐在 ON 子句中只比较两个表或结果集中的列。如果你要从底层表中筛选行，可在嵌入的 SELECT 语句的 WHERE 子句中再指定一个查找条件。

3. 在连接中嵌入连接

虽然通过连接两个表可解决很多问题，但很多时候都需要连接 3 个、4 个甚至更多的表，才能获取解决问题所需的所有数据。例如，你可能想通过一个查询取回所有有关菜品的信息：菜品的类型、菜品名以及制作菜品所需的所有食材。明白使用外连接可做什么后，你可能想列出所有的菜品类型（包括那些不包含任何菜品的菜品类型），还有有关菜品及其所需食材的详细信息。图 9-7 列出了为回答这个问题所需的所有表。

看起来好像需要 5 个表中的数据。与第 8 章一样，你可编写一个更复杂的 FROM 子句，在连接中嵌入连接。具体方法是：在每个可指定表名的地方，都可指定一个 JOIN 子句，并将其用括号括起。图 9-8 是两表连接的简化版本（省略了关联名并使用 ON 子句来实现简单的两表内连接或外连接）。

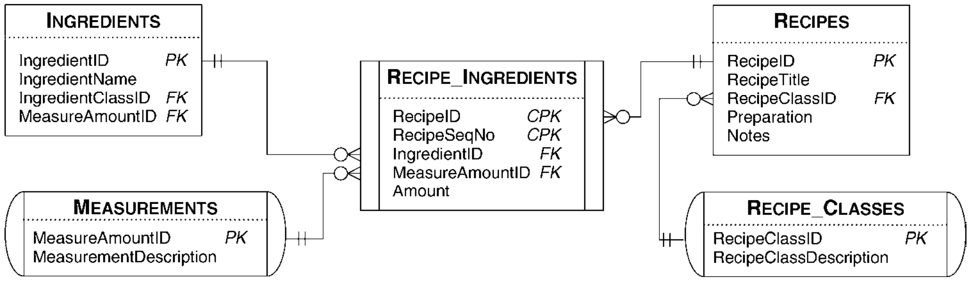


图 9-7　为取回有关菜品的所有信息，需要用到示例数据库 Recipes 中的哪些表

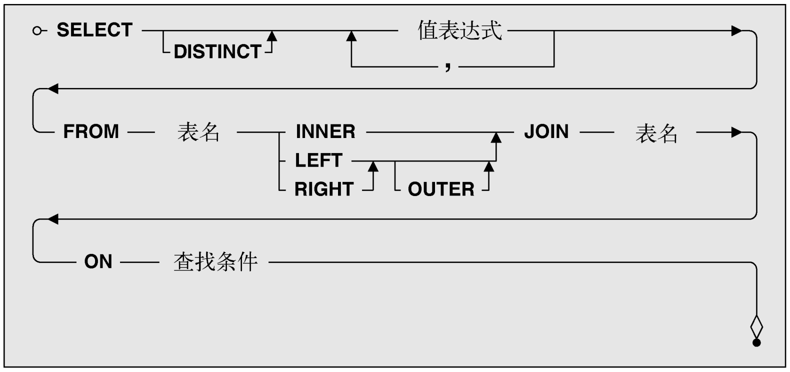


图 9-8　简单的两表连接

要添加第 3 个表，只需在第一个表名前加上左括号，在查找条件后面加上右括号，再依次插入关键字 JOIN、表名、关键字 ON 和查找条件，如图 9-9 所示。

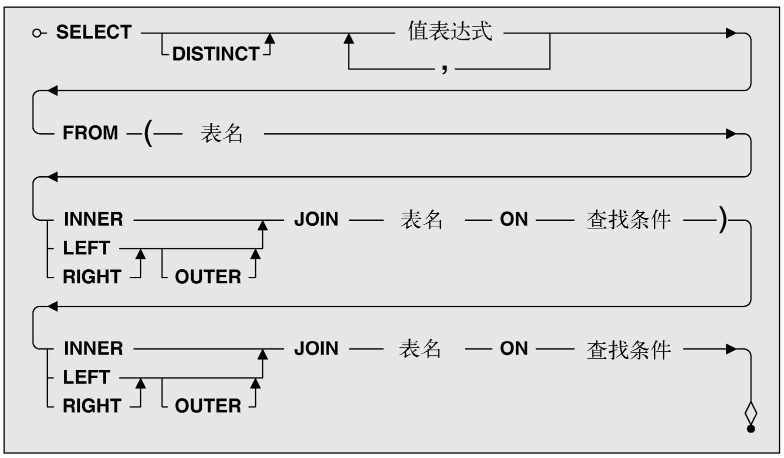


图 9-9　简单的三表连接

只要想一想就知道，括号内的两表连接是一个逻辑表，即内部结果集；这个结果集相当于取代了图 9-8 中的第一个表名。你可以不断重复这个过程（将整个 JOIN 子句放在括号内，再依次添加关键字 JOIN、表名、关键字 ON 和查找条件），直到获得所需的所有结果集。下面来发出一个请求，它需要从图 9-7 列出的所有表中获取数据（你可能使用这种请求来列出所有的菜品类型以及每种菜品类型中所有菜品的详细信息）。

“我需要从 Recipes 数据库中获取所有的菜品类型以及各类型中菜品的名称、制作说明、食材名、食材序号、食材数量和食材度量单位，并按菜品名和序号排序。”

转换	Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table left outer joined with the recipes table on recipe class ID in the recipe classes table matching recipe ID in the recipes table, then joined with the recipe ingredients table on recipe ID in the recipes table matching recipe ID in the recipe ingredients table, then joined with the ingredients table on ingredient ID in the ingredients table matching ingredient ID in the recipe ingredients table, and then finally joined with the measurements table on measurement amount ID in the measurements table matching measurement amount ID in the recipe ingredients table, order by recipe title and recipe sequence number（依次基于菜品类型 ID 相等将菜品类型表左外连接到菜品表、基于菜品 ID 相等内连接到菜品食材表、基于食材 ID 相等内连接到食材表、基于度量单位 ID 内连接到度量单位表，选择菜品类型描述、菜品名、制作说明、食材名、食材序号、食材数量和食材度量单位，并按菜品名和序号排序）
整理	Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description from ~~the~~ recipe classes ~~table~~ left outer join~~ed with the~~ recipes ~~table~~ on recipe_classes.recipe class ID ~~in the recipe classes table matching~~ = recipes.recipe class ID ~~in the recipes table, then~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matching~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table, then~~ inner join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matching~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table, and then finally~~ inner join~~ed with the~~ measurements ~~table~~ on measurements.measurement amount ID ~~in the measurements table matching~~ = recipe_ingredients.measurement amount ID ~~in the recipe ingredients table~~, order by recipe title, ~~and~~ recipe sequence number

SQL

```
SELECT Recipe_Cplasses.RecipeClassDescription,
    Recipes.RecipeTitle, Recipes.Preparation,
    Ingredients.IngredientName,
    Recipe_Ingredients.RecipeSeqNo,
    Recipe_Ingredients.Amount,
     Measurements.MeasurementDescription
FROM (((Recipe_Classes
LEFT OUTER JOIN Recipes
  ON Recipe_Classes.RecipeClassID =
     Recipes.RecipeClassID)
INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID)
INNER JOIN Measurements
  ON Measurements.MeasureAmountID =
     Recipe_Ingredients.MeasureAmountID
ORDER BY RecipeTitle, RecipeSeqNo
```

实际上，可将任何一个表名替换为两个连接。图 9-9 隐含这样一层意思，即必须先将第一个表与第二个表连接，再将结果与第三个表连接。但你也可先连接第二个和第三个表（只要第三个表与第二个表而不是第二个表相关），再将结果与第一个表连接，图 9-10 说明了这种方法。

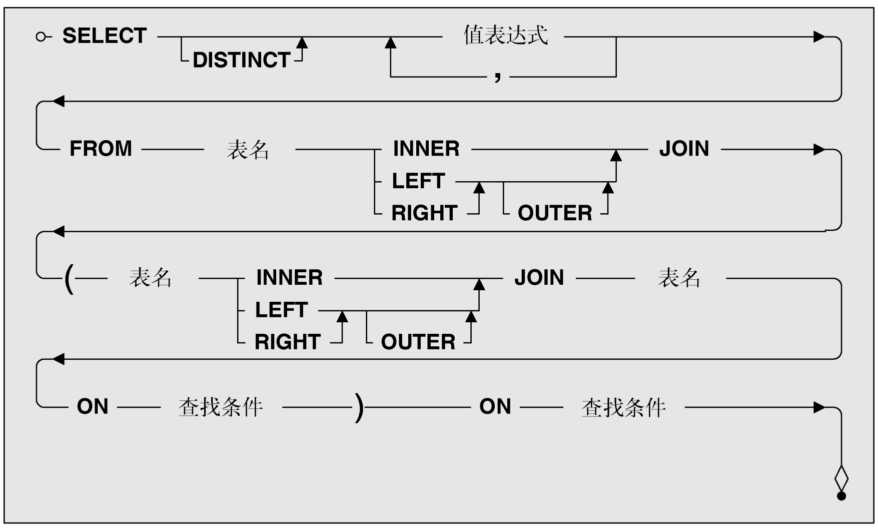


图 9-10　按另一种顺序连接两个以上的表

为了解决刚才这个需要用到 5 个表的问题，也可这样编写 SQL：

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
    Recipes.RecipeTitle, Recipes.Preparation,
    Ingredients.IngredientName,
    Recipe_Ingredients.RecipeSeqNo,
    Recipe_Ingredients.Amount,
    Measurements.MeasurementDescription
FROM Recipe_Classes LEFT OUTER JOIN
    (((Recipes
    INNER JOIN Recipe_Ingredients
    ON Recipes.RecipeID =
       Recipe_Ingredients.RecipeID)
    INNER JOIN Ingredients
    ON Ingredients.IngredientID =
       Recipe_Ingredients.IngredientID)
    INNER JOIN Measurements
    ON Measurements.MeasureAmountID =
       Recipe_Ingredients.MeasureAmountID)
ON Recipe_Classes.RecipeClassID =
    Recipes.RecipeClassID
ORDER BY RecipeTitle, RecipeSeqNo
```

别忘了，有些数据库系统的优化器对连接的顺序很敏感。如果你的查询包含很多连接，且在大型数据库中执行需要的时间很长，通过调整 SQL 语句中连接的顺序，也许可以提高运行速度。

你可能注意到了，在前面的多连接示例中，只使用了一个外连接。你可能会问：在复杂的连接中，是否可以使用多个外连接？这样做有意义吗？我们假设不仅有一些菜品类型不包含任何菜品，还有一些菜品不包含任何已定义的食材。在前面的示例中，对于 Recipes 表中的行，如果 Recipe_Ingredients 表中没有任何行与之匹配，它将不会出现在结果集中，因为内连接会将它们排除。下面来请求列出所有的菜品。

> “我需要从数据库 Recipes 中获取所有菜品类型、所有菜品名以及制作说明、食材名、食材序号、食材数量和食材度量单位，并按菜品名和序号排序。”

转换

Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table left outer joined with the recipes table on recipe class ID in the recipe classes table matching recipe class ID in the recipes table, then left outer joined with the recipe ingredients table on recipe ID in the recipes table matching recipe ID in the recipe ingredients table, then joined with the ingredients table on ingredient ID in the ingredients table matching ingredient ID in the recipe ingredients table, and then finally joined with the measurements table on measurement amount ID in the measurements table matching measurement amount ID in the recipe ingredients table, order by recipe title and recipe sequence number（依次基于菜品类型 ID 相等将菜品类型表左外连接到菜品表、基于菜品 ID 相等左外连接到菜品食材表、基于食材 ID 相等内连接到食材表、基于度量单位 ID 内连接到度量单位表，选择菜品类型描述、菜品名、制作说明、食材名、食材序号、食材数量和食材度量单位，并按菜品名和序号排序）

整理

Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description from ~~the~~ recipe classes ~~table~~ left outer join~~ed with the~~ recipes ~~table~~ on recipe_classes.recipe class ID ~~in the recipe classes table matching~~ = recipes.recipe class ID ~~in the recipes table, then~~ left outer join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matching~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table, then~~ inner join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matching~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table, and then finally~~ inner join~~ed with the~~ measurements ~~table~~ on measurements.measurement amount ID ~~in the measurements table matching~~ = recipe_ingredients.measurement amount ID ~~in the recipe ingredients table~~, order by recipe title ~~and~~ recipe sequence number

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
    Recipes.RecipeTitle, Recipes.Preparation,
    Ingredients.IngredientName,
    Recipe_Ingredients.RecipeSeqNo,
    Recipe_Ingredients.Amount,
    Measurements.MeasurementDescription
FROM (((Recipe_Classes
LEFT OUTER JOIN Recipes
  ON Recipe_Classes.RecipeClassID =
     Recipes.RecipeClassID)
LEFT OUTER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
```

```
INNER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID)
INNER JOIN Measurements
  ON Measurements.MeasureAmountID =
     Recipe_Ingredients.MeasureAmountID
ORDER BY RecipeTitle, RecipeSeqNo
```

千万要小心！这种多外连接仅在沿一对多关系路径进行时才会按预期那样工作。再来看看 Recipe_Classes、Recipes 和 Recipe_Ingredients 表之间的关系，如图 9-11 所示。

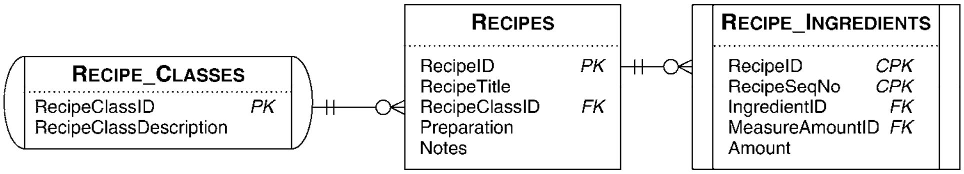


图 9-11 Recipe_Classes 表、Recipes 表和 Recipe_ Ingredients 表之间的关系

一对多关系有时也被称为**父子关系**：每个父行（位于关系的“一”端）都有零个或更多个子行（位于关系的“多”端）。除非“多”端存在孤行（如在 Recipes 表中，RecipeClassID 列为 Null 的行），否则子表中的每行在父表中都有与之匹配的行。因此将 Recipe_Classes 表左连接到 Recipes 表是有意义的，它选择表 Recipe_Classes 中这样的父行，即在 Recipes 表中没有与之匹配的子行；而将 Recipe_Classes 表右连接到 Recipes 表得到的结果将与内连接相同（将所有孤行排除在外）。

同理，将 Recipes 表左连接到 Recipe_Ingredients 表也是有意义的，因为有些菜品包含的所有食材可能还未输入。将 Recipes 表右连接到 Recipe_Ingredients 不管用，因为在 Recipe_Ingredients 表中，连接列（RecipeID）是复合主键的一部分。因此 Recipe_Ingredients 表中不可能有孤行，因为主键列的值不能为 Null。

下面更进一步，同时取回所有的食材，包括那些未出现在任何菜品中的食材。首先，来仔细看看包括 Ingredients 在内的各个表之间的关系，如图 9-12 所示。

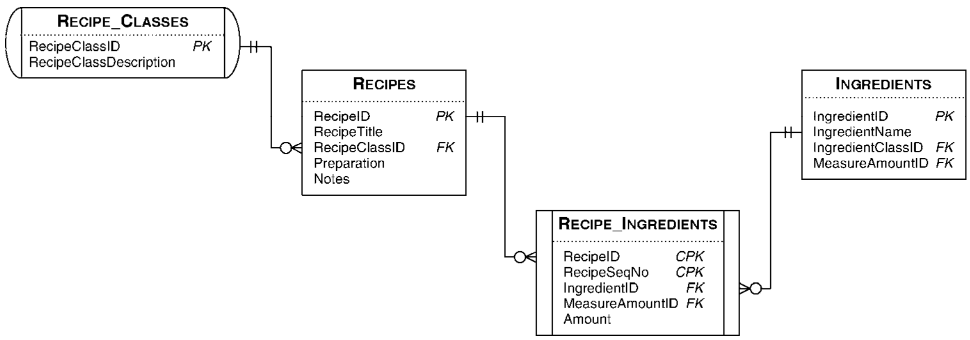


图 9-12 Recipe_Classes 表、Recipes 表、Recipe_Ingredients 表和 Ingredients 表之间的关系

我们来尝试解决这个问题（请注意这里会有陷阱！）

> “我需要从数据库 Recipes 中获取所有菜品类型、所有菜品名以及制作说明、食材序号、食材数量和食材度量单位，还有所有的食材名，并按菜品名和序号排序。”

转换	Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table left outer joined with the recipes table on recipe class ID in the recipe classes table matches class ID in the recipes table, then left outer joined with the recipe ingredients table on recipe ID in the recipes table matches recipe ID in the recipe ingredients table, then joined with the measurements table on measurement amount ID in the measurements table matches measurement amount ID in the measurements table, and then finally right outer joined with the ingredients table on ingredient ID in the ingredients table matches ingredient ID in the recipe ingredients table, order by recipe

	title and recipe sequence number（依次基于菜品类型 ID 相等将菜品类型表左外连接到菜品表、基于菜品 ID 相等左外连接到菜品食材表、基于度量单位 ID 内连接到度量单位表、基于食材 ID 相等右外连接到食材表，选择菜品类型描述、菜品名、制作说明、食材名、食材序号、食材数量和食材度量单位，并按菜品名和序号排序）
整理	Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description from ~~the~~ recipe classes ~~table~~ left outer join~~ed with the~~ recipes ~~table~~ on recipe_classes.recipe class ID ~~in the recipe classes table matches~~ = recipes.class ID ~~in the recipes table, then~~ left outer join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matches~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table, then~~ inner join~~ed with the~~ measurements ~~table~~ on measurements.measurement amount ID ~~in the measurements table matches~~ = measurements.measurement amount ID ~~in the measurements table, and then finally~~ right outer join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~, order by recipe title, ~~and~~ recipe sequence number

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
    Recipes.RecipeTitle, Recipes.Preparation,
    Ingredients.IngredientName,
    Recipe_Ingredients.RecipeSeqNo,
    Recipe_Ingredients.Amount,
    Measurements.MeasurementDescription
FROM (((Recipe_Classes
LEFT OUTER JOIN Recipes
  ON Recipe_Classes.RecipeClassID =
     Recipes.RecipeClassID)
LEFT OUTER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
INNER JOIN Measurements
  ON Measurements.MeasureAmountID =
     Recipe_Ingredients.MeasureAmountID)
RIGHT OUTER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID
ORDER BY RecipeTitle, RecipeSeqNo
```

你认为这管用吗？实际上，答案绝对是否定的！大多数数据库系统先分析整个 FROM 子句，再尝试确定连接表的最有效方式。然而，假设数据库系统完全按我们使用括号指定的顺序连接，这意味着数据库系统将从内往外依次连接（最先将 Recipe_Classes 表连接到 Recipes 表）。

由于 Recipe_Classes 表中的有些行在 Recipes 表中没有任何匹配的行，因此第一个连接将返回 Recipes.RecipeClassID 列的值为 Null 的行。回过头去看图 9-12，将发现 Recipe_Classes 和 Recipes 之间为一对多关系，因此除非有些菜品没有指定菜品类型，否则将返回 Recipes 表中的所有行。下一个到 Recipe_Ingredients 表的连接也是左外连接，即要求获取前一个连接（Recipe_Classes 到 Recipes 的连接）得到的结果集中所有的行（而不考虑 Null 值），还有 Recipe_Ingredients 表中匹配的行。同样，由于 Recipe_Classes 表中的有些行可能在 Recipes 表中没有匹配的行，或者 Recipes 表中的有些行在 Recipe_Ingredients 表中没有匹配的行，因此在从 Recipe_Ingredients 表中取回的行中，有些行的 IngredientID 列的值可能为 Null。这里执行的两个连接操作都是沿一对多关系进行的：从 Recipe_Classes 到 Recipes，再从 Recipes 到 Recipe_Ingredients。到目前为止，一切顺利（顺便说一句，最后的到 Measurements 表的内连接没问题，因为所有食材都有合法的度量单位 ID）。

接下来麻烦来了。最后的右外连接获取 Ingredients 表中所有的行，还有之前的连接结果中匹配的行。第 5 章说过，Null 是个非常特殊的值，它不与其他任何值相等，包括另一个 Null。当我要求取回 Ingredients 表中的所有行时，这些行的 IngredientID 都不是 Null。在之前的连接结果中，所有 IngredientID 为 Null 的行都是不匹配的，因此最后的连接将把它们排除在外。换句话说，你将看到没有在任何菜品中使用的食材，但看不到不包含任何菜品的菜品类型，也看不到不包含任何已定义食材的菜品。

如果你使用的数据库系统在执行这个查询时调整了连接的顺序，你可能看到不包含任何菜品的菜品类型以及不包含任何已定义食材的菜品，但将看不到未在任何菜品中使用的食材，这都是 Null 匹配问题导致的。有些数据库系统可能会发现这种逻辑问题，进而拒绝执行这个查询，并显示类似于“模糊的外连接”这样的错误消息。之所以会出现这种问题，是因为你沿多对一关系的方向执行外连接。下山很容易，但要上山得有特殊工具。这种问题如何解决呢？要找到答案，请接着阅读下一节。

9.3 全外连接

全外连接（FULL OUTER JOIN）既不是左连接，也不是右连接，而是左右连接。它返回连接涉及的两个表或结果集中的所有行。对于左表中的行，如果没有与之匹配的行，则在结果集中，右表各列的值为 Null；相反，对于右表中的行，如果没有与之匹配的行，则在结果集中，左表各列的值为 Null。

9.3.1 语法

至此，你使用连接有一段时间了，因此很容易理解全外连接的语法。图 9-13 显示了全外连接的语法图。

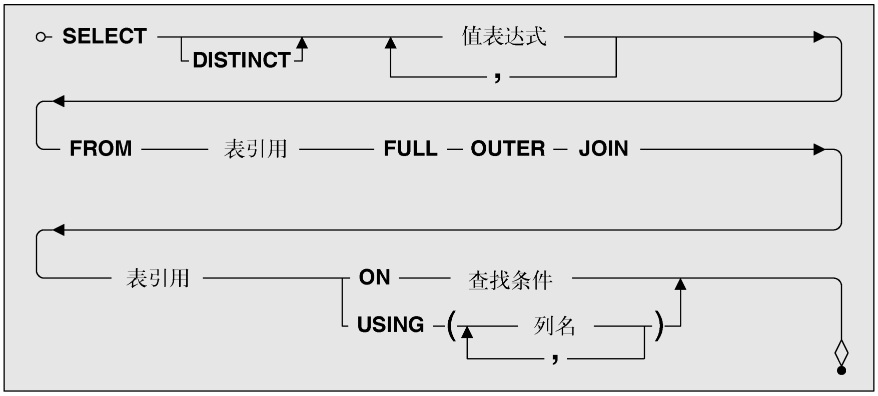


图 9-13 全外连接的语法图

为简单起见，我使用术语**表引用**作为表名、SELECT 语句和连接结果的统称。再来看看前一节末尾引入的问题。现在可以使用全外连接正确地解决它了。

> “我需要从数据库 Recipes 中获取所有菜品类型、所有菜品名以及制作说明、食材序号、食材数量和食材度量单位，还有所有的食材名，并按菜品名和序号排序。”

转换	Select the recipe class description, recipe title, preparation instructions, ingredient name, recipe sequence number, amount, and measurement description from the recipe classes table full outer joined with the recipes table on recipe class ID in the recipe classes table matches recipe class ID in the recipes table, then left outer joined with the recipe ingredients table on recipe ID in the recipes table matches recipe ID in the recipe ingredients table, then joined with the measurements table on measurement amount ID in the measurements table matches measurement amount ID in the recipe ingredients table, and then finally full outer joined with the ingredients table on ingredient ID in the ingredients table matches ingredient ID in the recipe ingredients table, order by recipe title and recipe sequence number（依次基于菜品类型 ID 相等将菜品类型表全外连接到菜品表、基于菜品 ID 相等左外连接到菜品食材表、基于度量单位 ID 内连接到度量单位表、基于食材 ID 相等全外连接到食材表，选择菜品类型描述、菜品名、制作说明、食材名、食材序号、食材数量和食材度量单位，并按菜品名和序号排序）
整理	Select ~~the~~ recipe class description, recipe title, preparation ~~instructions~~, ingredient name, recipe sequence number, amount, ~~and~~ measurement description from ~~the~~ recipe classes ~~table~~ full outer join~~ed with the~~ recipes ~~table~~ on recipe_classes.recipe class ID ~~in the recipe classes table matches~~ = recipes.recipe class ID ~~in the recipes table, then~~ left outer join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matches~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table, then~~ inner join~~ed with the~~ measurements ~~table~~ on measurements.measurement amount ID ~~in the measurements table matches~~ = recipe_ingredients.measurement amount ID ~~in the recipe ingredients table, and then finally~~ full outer join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~, order by recipe title ~~and~~ recipe sequence number

SQL	`SELECT Recipe_Classes.RecipeClassDescription,`
	`Recipes.RecipeTitle, Recipes.Preparation,`
	`Ingredients.IngredientName,`
	`Recipe_Ingredients.RecipeSeqNo,`
	`Recipe_Ingredients.Amount,`
	`Measurements.MeasurementDescription`
	`FROM (((Recipe_Classes`
	`FULL OUTER JOIN Recipes`

```
            ON Recipe_Classes.RecipeClassID =
               Recipes.RecipeClassID)
            LEFT OUTER JOIN Recipe_Ingredients
            ON Recipes.RecipeID =
               Recipe_Ingredients.RecipeID)
            INNER JOIN Measurements
            ON Measurements.MeasureAmountID =
               Recipe_Ingredients.MeasureAmountID)
            FULL OUTER JOIN Ingredients
            ON Ingredients.IngredientID =
               Recipe_Ingredients.IngredientID
         ORDER BY RecipeTitle, RecipeSeqNo
```

第一个和最后一个连接取回两边的所有行，这解决了 Null 不与任何值匹配的问题。现在，你不仅能看到不包含任何菜品的菜品类型以及不包含任何已定义食材的菜品，还能看到未在任何菜品中使用的食材。将第一个连接替换为左外连接可能不会有任何问题，但鉴于你无法预测数据库系统将如何处理嵌套连接，因此在开头和末尾都应使用全外连接，以确保答案是正确的。

❖ **说明**　你可能猜到了，不支持 SQL 标准定义的左/右外连接语法的数据库系统也提供了特殊的全外连接语法。有关你使用的数据库系统具体使用什么样的非标准语法来定义外连接，请参阅其文档。例如，较早的 Microsoft SQL Server 版本支持如下语法（请注意 WHERE 子句中的星号）：

```
SELECT Recipe_Classes.RecipeClassDescription,
    Recipes.RecipeTitle
FROM Recipe_Classes, Recipes
WHERE Recipe_Classes.RecipeClassID *=*
    Recipes.RecipeClassID
```

在不支持任何全外连接但支持左/右外连接的数据库系统中，可对左外连接和右外连接的结果执行 UNION 操作来获得与全外连接相同的结果。UNION 将在下一章更详细地讨论。由于使用 WHERE 子句来定义全外连接的语法因产品而异，因此如果你需要使用多种非标准产品，可能就需要学习多种不同的语法。

9.3.2　基于非键值的全外连接

前面讨论的都是使用基于相关键值的外连接来连接表或结果集，但使用基于非键值的外连接可解决一些有趣的问题。例如，前一章演示了如何在数据库 School Scheduling 中找出名字相同的学生和教员。现在假设要列出所有教员和所有学生，并指出哪些教员有同名的学生、哪些教员没有同名的学生、哪些学生有同名的教员以及哪些学生没有同名的教员。为此，可使用全外连接。

“显示所有的学生和所有的教员，并将同名的学生和教员列在一起。”

转换	Select student full name and staff full name from the students table full outer joined with the staff table on first name in the students table matches first name in the staff table（基于名字相同将学生表全外连接到教员表，并选择学生的姓名和教员的姓名）
整理	Select student full name ~~and~~ staff full name from ~~the~~ students ~~table~~ full outer join~~ed with the~~ staff ~~table~~ on students.first name ~~in the students table matches~~ = staff.first name ~~in the staff table~~
SQL	SELECT (Students.StudFirstName || ' ' || Students.StudLastName) AS StudFullName, (Staff.StfFirstName || ' ' || Staff.StfLastName) AS StfFullName FROM Students FULL OUTER JOIN Staff ON Students.StudFirstName = Staff.StfFirstName

9.3.3　UNION JOIN

如果不提一提 UNION JOIN，对外连接的讨论就不完整。在 SQL 标准中，UNION JOIN 是将匹配行排除的全外连接，其语法如图 9-14 所示。

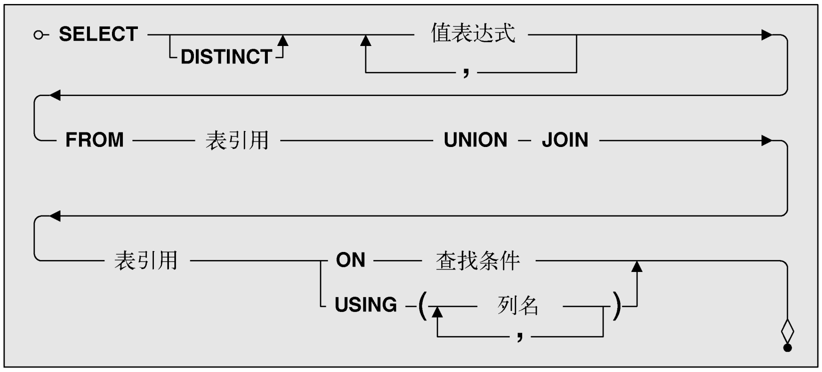


图 9-14　UNION JOIN 的 SQL 语法

你可能猜到了，支持 UNION JOIN 的商业实现不多。坦率地说，我很难想到使用 UNION JOIN 的理由。

9.4　外连接的用途

外连接让你不仅能够获取匹配的行，还能获取不匹配的行，因此非常适合用于确定一个表中的哪些行在另一个表中没有匹配的行；它还帮助你确定哪些行与多行匹配，但不与所有行匹配。另外，外连接还可用于创建列出所有类别（不管它们是否与其他表中的行匹配）或所有顾客（不管它们是否下了订单）的报表。下面列出几种可使用外连接来解决的问题。

9.4.1　查找缺失值

有时候，你只想找出缺失值，为此可结合使用外连接和 Null 测试。下面列举了一些可使用外连接来解决的“值缺失”问题。

“哪些商品从未被订购过。”
“告诉我哪些顾客从未订购过头盔。”
“列出未签订任何演出合约的演唱组合。”
“显示未签订任何演出合约的经纪人。”
“指出还未举行的联赛。”
“列出不讲授课程的教员。”
“显示未退过课的学生。”
“告知没有学生注册的课程。”
“列出还未被任何菜品使用的食材。”
“显示不包含任何菜品的菜品类型。”

9.4.2　查找部分匹配信息

如果能够列出一个或多个表中所有的行以及相关表中的匹配行，将很有用，对撰写报告来说尤其如此。下面是一些可使用外连接来解决的“部分匹配”问题。

“列出所有的商品以及包含各件商品的订单的日期。”
“显示所有的顾客及其自行车订单。”
“列出所有的音乐风格以及喜欢各种风格的顾客。”
“列出所有的演唱组合及其签订的演出合约。”
“列出所有的投球手及其得分超过 160 的比赛局次。”
“显示所有的联赛及其已举行的比赛场次。”
“告知所有的科目类别及其包含的课程。”
“列出所有的学生及其注册的课程。”
“显示所有的教员及其讲授的课程。”
“列出所有的菜品类型、所有的菜品及其使用的食材。”
“显示所有的食材及其被用来制作的菜品。”

9.5 语句举例

至此，你明白了使用外连接编写查询的机制，还知道了一些可使用外连接来解决的问题类型。下面来看一些外连接使用示例，这些示例都摘自示例数据库，演示了如何使用外连接来查找缺失值或部分匹配的值。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库，查询名以 CH09 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明**　这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。

- **Sales Orders 数据库**

“哪些商品从未被订购过？”

转换/整理	Select product number ~~and~~ product name from ~~the~~ products ~~table~~ left outer join~~ed with the~~ order details ~~table~~ on products.product number ~~in the products table matches~~ = order_details.product number ~~in the order details table~~ where ~~the~~ order detail order number is null（基于商品编号将商品表左外连接到订单详情表，并选择订单号为 Null 的商品的编号和名称）

SQL

```
SELECT Products.ProductNumber,
    Products.ProductName
FROM Products
LEFT OUTER JOIN Order_Details
  ON Products.ProductNumber =
     Order_Details.ProductNumber
WHERE Order_Details.OrderNumber IS NULL
```

CH09_Products_Never_Ordered（2 行）

ProductNumber	ProductName
4	Victoria Pro All Weather Tires
23	Ultra-Pro Rain Jacket

“显示所有顾客及其自行车订单。”

第 1 次转换	Select customer full name, order date, product name, quantity ordered, and quoted price from the customers table left outer joined with the orders table on customer ID, then joined with the order details table on order number, then joined with the products table on product number, then finally joined with the categories table on category ID where category description is “Bikes”（依次基于顾客 ID 将顾客表左外连接到订单表、基于订单号内连接到订单详情表、基于商品编号内连接到商品表、基于类别 ID 内连接到类别表，并从类别描述为 Bikes 的行中选择顾客的姓名、订单日期、商品名、订购的数量以及报价）

第2次转换/整理

Select customer full name, order date, product name, quantity ordered, ~~and~~ quoted price from ~~the~~ customers ~~table~~ left outer join~~ed with~~ (Select customer ID, order date, product name, quantity ordered, ~~and~~ quoted price from ~~the~~ orders ~~table~~ inner join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table matches~~ = order_details.order number ~~in the order details table, then~~ join~~ed with the~~ products ~~table~~ on order_details.product number ~~in the order details table matches~~ = products.product number ~~in the products table, then finally~~ join~~ed with the~~ categories ~~table~~ on categories.category ID ~~in the categories table matches =~~ products.category ID ~~in the products table~~ where category description ~~is~~ = 'Bikes') as rd on customers.customer ID ~~in the customers table matches~~ = rd.customerID ~~in the embedded SELECT statement~~（依次基于订单号相同将订单表内连接到订单详情表、基于商品编号相同内连接到商品表、基于类别ID相同内连接到类别表，从类别描述为Bikes的行中选择顾客ID、订单日期、商品名、订购数量和报价，并将这个结果集命名为rd；基于顾客ID相同将顾客表左外连接到上述结果集，并选择顾客姓名、订单日期、商品名、订购数量和报价）

❖ **说明** 由于要查找特定的订单（自行车订单），我将转换过程分成了两步，以指出需要先筛选订单，再执行外连接。

SQL

```
SELECT Customers.CustFirstName || ' ' ||
    Customers.CustLastName AS CustFullName,
    RD.OrderDate, RD.ProductName,
   RD.QuantityOrdered, RD.QuotedPrice
FROM Customers
LEFT OUTER JOIN
   (SELECT Orders.CustomerID, Orders.OrderDate,
       Products.ProductName,
       Order_Details.QuantityOrdered,
       Order_Details.QuotedPrice FROM ((Orders
    INNER JOIN Order_Details
       ON Orders.OrderNumber =
          Order_Details.OrderNumber)
    INNER JOIN Products
       ON Order_Details.ProductNumber =
          Products.ProductNumber)
    INNER JOIN Categories
       ON Categories.CategoryID =
          Products.CategoryID
WHERE Categories.CategoryDescription =
        'Bikes')
   AS RD
ON Customers.CustomerID = RD.CustomerID
```

❖ **说明** 这个请求确实很棘手，因为要列出所有的顾客，同时通过使用外连接只列出自行车订单。如果基于第1次转换结果编写SQL，将不会列出没有订购自行车的顾客。将Customers表外连接到Orders表将返回所有顾客和所有订单；使用筛选器只选择自行车订单后，将只返回订购了自行车的顾客。第2次转换指出了如何正确地解决这个问题：先创建一个只包含自行车订单的内部结果集，再将Customers表外连接到这个结果集，从而得到最终的答案。

CH09_All_Customers_And_Any_Bike_Orders（914行）

CustFullName	OrderDate	ProductName	Quantity Ordered	QuotedPrice
Suzanne Viescas				
William Thompson	2017-12-24	Trek 9000 Mountain Bike	5	$1,164.00
William Thompson	2018-01-16	Trek 9000 Mountain Bike	6	$1,164.00
William Thompson	2017-10-12	Viscount Mountain Bike	2	$635.00
William Thompson	2017-10-06	Viscount Mountain Bike	5	$615.95
William Thompson	2018-01-16	Trek 9000 Mountain Bike	4	$1,200.00
William Thompson	2017-10-12	Trek 9000 Mountain Bike	3	$1,200.00
William Thompson	2018-01-08	Trek 9000 Mountain Bike	2	$1,200.00
		<< 其他行 >>		

（看起来 William Thompson 真是大主顾！）

- **Entertainment Agency 数据库**

“列出从未签约的演唱组合。”

转换/整理　Select entertainer ID ~~and~~ entertainer stage name from ~~the~~ entertainers ~~table~~ left outer join~~ed with the~~ engagements ~~table~~ on entertainers.entertainer ID ~~in the entertainers table matches~~ = engagements.entertainer ID ~~in the engagements table~~ where engagement number is null（基于演唱组合 ID 相等将演唱组合表左外连接到演出合约表，从演出合约编号为 Null 的行中选择演唱组合 ID 和艺名）

SQL

```
SELECT Entertainers.EntertainerID,
    Entertainers.EntStageName
FROM Entertainers
LEFT OUTER JOIN Engagements
  ON Entertainers.EntertainerID =
     Engagements.EntertainerID
WHERE Engagements.EngagementNumber IS NULL
```

CH09_Entertainers_Never_Booked（1 行）

EntertainerID	EntStageName
1009	Katherine Ehrlich

“列出所有的音乐风格以及喜欢各种风格的顾客。”

转换/整理　Select style ID, style name, customer ID, customer first name, ~~and~~ customer last name from ~~the~~ musical styles ~~table~~ left outer join~~ed with~~ (~~the~~ musical preferences ~~table~~ inner join~~ed with the~~ customers ~~table~~ on musical_preferences.customer ID ~~in the musical preferences table matches~~ = customers.customer ID ~~in the customers table~~) on musical_styles.style ID ~~in the musical styles table matches~~ = musical_preferences.style ID ~~in the musical preferences table~~（基于顾客 ID 相同将音乐喜好表内连接到顾客表，再基于风格 ID 相同将音乐风格表左外连接到上述内连接结果，并选择音乐风格的 ID 和名称以及顾客的名和姓）

SQL

```
SELECT Musical_Styles.StyleID,
    Musical_Styles.StyleName,
    Customers.CustomerID,
    Customers.CustFirstName,
    Customers.CustLastName
FROM Musical_Styles
LEFT OUTER JOIN (Musical_Preferences
    INNER JOIN Customers
    ON Musical_Preferences.CustomerID =
       Customers.CustomerID)
    ON Musical_Styles.StyleID =
       Musical_Preferences.StyleID
```

CH09_All_Styles_And_Any_Customers（41 行）

StyleID	StyleName	CustomerID	CustFirstName	CustLastName
1	40 s Ballroom Music	10015	Carol	Viescas
1	40 s Ballroom Music	10011	Joyce	Bonnicksen
2	50s Music			
3	60s Musicb	10002	Deb	Waldal
4	70s Music	10007	Liz	Keyser
5	80s Music	10014	Mark	Rosales
6	Country	10009	Sarah	Thompson
7	Classical	10005	Elizabeth	Hallmark
		<< 其他行 >>		

（看起来没人喜欢 20 世纪 50 年代的音乐！）

❖ **说明**　我非常小心地编写 FROM 子句，让数据库系统先将 Musical_Preferences 表内连接到 Customers 表，再将 Musical_Styles 表外连接到上述内连接的结果。如果你使用的数据库系统按从左到右的顺序处理连接，可能必须在 FROM 子句中先指定内连接，再右外连接到 Musical_Styles 表。在 Microsoft Office Access 中，必须使用一条嵌入的 SELECT 语句来执行内连接，这样才能得到正确的结果。

- **School Scheduling 数据库**

“列出没有讲授任何课程的教员。”

转换/整理　Select staff first name ~~and~~ staff last name from ~~the~~ staff ~~table~~ left outer join~~ed with the~~ faculty classes ~~table~~ on staff.staff ID ~~in the staff table matches~~ = faculty_classes.staff ID ~~in the faculty classes table~~ where class ID is null（基于教员 ID 相同将教员表左外连接到教员课程表，并从课程 ID 为 Null 的行中选择教员的名和姓）

SQL
```
SELECT Staff.StfFirstName, Staff.StfLastName,
FROM Staff LEFT OUTER JOIN Faculty_Classes
  ON Staff.StaffID = Faculty_Classes.StaffID
WHERE Faculty_Classes.ClassID IS NUL
```

CH09_Staff_Not_Teaching（5 行）

StfFirstName	StfLastName
Jeffrey	Smith
Tim	Smith
Kathryn	Patterson
Joe	Rosales III
Carolyn	Coie

“显示从未退过课的学生。”

转换/整理　Select student full name from ~~the~~ students ~~table~~ left outer join~~ed with~~ (Select student ID from ~~the~~ student schedules ~~table~~ inner join~~ed with the~~ student class status ~~table~~ on student_class_status.class status ~~in the student class status table matches~~ = student_schedules.class status ~~in the student schedules table~~ where class status description ~~is~~ = ‘withdrew’) as withdrew on students.student ID ~~in the students table matches~~ = withdrew.student ID ~~in the embeddedd SELECT statement~~ where ~~the~~ student_schedules.student ID ~~in the student schedules table~~ is null（基于课程状态相同将学生选课表内连接到学生课程状态表，再从课程状态描述为 withdrew 的行中选择学生 ID，并将这个结果集命名为 withdrew；基于学生 ID 相同将学生表左外连接到该结果集，并从学生 ID 为 Null 的行中选择学生姓名）

SQL
```
SELECT Students.StudLastName || ', ' ||
    Students.StudFirstName AS StudFullName
FROM Students
LEFT OUTER JOIN
    (SELECT Student_Schedules.StudentID
    FROM Student_Class_Status
    INNER JOIN Student_Schedules
      ON Student_Class_Status.ClassStatus =
         Student_Schedules.ClassStatus
    WHERE Student_Class_Status.ClassStatus
      Description = 'withdrew')
   AS Withdrew
ON Students.StudentID = Withdrew.StudentID
WHERE Withdrew.StudentID IS NULL
```

❖ **说明**　在这个示例中，也必须在嵌入的 SELECT 语句中使用筛选器。如果在主查询的 WHERE 子句中使用筛选器，将不会有任何结果。别忘了，需要在左连接的右端（或右连接的左端）使用筛选器时，必须在嵌入的 SELECT 语句中这样做。

CH09_Students_Never_Withdrawn（16 行）

StudFullName
Patterson, Kerry
Stadick, Betsy
Galvin, Janice
Hartwig, Doris
Bishop, Scott
Hallmark, Elizabeth
Sheskey, Sara
Smith, Karen
<< 其他行 >>

“列出所有的科目类别及其所有课程。”

转换/整理

Select category description, subject name, classroom ID, start date, start time, ~~and~~ duration from ~~the~~ categories ~~table~~ left outer join~~ed with the~~ subjects ~~table~~ on categories.category ID ~~in the categories table matches~~ = subjects.category ID ~~in the subjects table, then~~ left outer ~~joined with the~~ classes ~~table~~ on subjects.subject ID ~~in the subjects table matches~~ = classes.subject ID ~~in the classes table~~（依次基于类别 ID 相同将类别表左外连接到科目表、基于科目 ID 相同左外连接到课程表，并选择类别描述、科目名、教室 ID、开始日期、开始时间和时长）

SQL

```
SELECT Categories.CategoryDescription,
    Subjects.SubjectName, Classes.ClassroomID,
    Classes.StartDate, Classes.StartTime,
    Classes.Duration
FROM (Categories
LEFT OUTER JOIN Subjects
  ON Categories.CategoryID = Subjects.
CategoryID)
LEFT OUTER JOIN Classes
  ON Subjects.SubjectID = Classes.SubjectID
```

❖ **说明**　这里也非常小心地指定了连接顺序，才确保得到期望的答案。

CH09_All_Categories_All_Subjects_Any_Classes（145 行）

Category Description	SubjectName	ClassroomID	StartDate	StartTime	Duration
Accounting	Financial Accounting Fundamentals I	3305	2017-09-11	16:00	50
Accounting	Financial Accounting Fundamentals I	3305	2018-01-15	16:00	50
Accounting	Financial Accounting Fundamentals II	3307	2017-09-12	13:00	80
Accounting	Fundamentals of Managerial Accounting	3307	2018-01-16	13:00	80
Accounting	Intermediate Accounting				
Accounting	Business Tax Accounting				
Art	Introduction to Art	1231	2017-09-12	10:00	50
Art	Introduction to Art	1231	2018-01-16	10:00	50
<< 其他行 >>					

如果你查看完整的结果集，将发现没有为科目 Introduction to Business、Developing a Feasibility Plan、Introduction to Entrepreneurship 和 Information Technology I and II 安排任何课程；另外，类别 Psychology、French 和 German 没有包含任何科目。

- **Bowling League 数据库**

“列出还未举行的联赛。”

转换/整理

Select tourney ID, tourney date, ~~and~~ tourney location from ~~the~~ tournaments ~~table~~ left outer join~~ed with the~~ tourney matches ~~table~~ on tournaments.tourney ID ~~in the tourna ments table matches~~ = tourney_matches.tourney ID ~~in the tourney matches table~~ where match ID is null（基于联赛 ID 相同将联赛表左外连接到联赛场次表，并从场次 ID 为 Null 的行中选择联赛 ID、联赛日期和联赛地点）

SQL

```
SELECT Tournaments.TourneyID,
    Tournaments.TourneyDate,
    Tournaments.TourneyLocation
FROM Tournaments
LEFT OUTER JOIN Tourney_Matches
  ON Tournaments.TourneyID =
     Tourney_Matches.TourneyID
WHERE Tourney_Matches.MatchID IS NULL
```

CH09_Tourney_Not_Yet_Played（6 行）

TourneyID	TourneyDate	TourneyLocation
15	2018-07-12	Red Rooster Lanes
16	2018-07-19	Thunderbird Lanes
17	2018-07-26	Bolero Lanes
18	2018-08-02	Sports World Lanes
19	2018-08-09	Imperial Lanes
20	2018-08-16	Totem Lanes

“列出所有的投球手及其得分超过 180 的比赛场次。”

第 1 次转换

Select bowler name, tourney date, tourney location, match ID, and raw score from the bowlers table left outer joined with the bowler scores table on bowler ID, then inner joined with the tourney matches table on match ID, then finally inner joined with the tournaments table on tournament ID where raw score in the bowler scores table is greater than 180（依次基于投球手 ID 将投球手表左外连接到投球手得分表、基于场次 ID 内连接到联赛场次表、基于联赛 ID 内连接到联赛表，再从原始得分超过 180 的行中选择投球手姓名、联赛日期、联赛地点、场次 ID 和原始得分）

上述转换不管用，你明白其中的原因吗？需要对一个位于左连接右边的表进行筛选，因此需要将该筛选器放在一条嵌入的 SELECT 语句中。下面来重新转换并进行整理，再解决这个问题。

第 2 次转换/整理

Select bowler name, tourney date, tourney location, match ID, ~~and~~ raw score from ~~the~~ bowlers ~~table~~ left outer join~~ed with~~ (Select tourney date, tourney location, match ID, bowler ID, ~~and~~ raw score from ~~the~~ bowler scores ~~table~~ inner join~~ed with the~~ tourney matches ~~table~~ on bowler_scores.match ID ~~in the bowler scores table matches~~ = tourney_matches.match ID ~~in the tourney matches table, then~~ inner join~~ed with the~~ tournaments ~~table~~ on tournaments.tournament ID ~~in the tournaments table matches~~ = tourney_matches.tournament ID ~~in the tourney matches table~~ where raw score ~~is greater than~~ > 180) as ti on bowlers.bowler ID ~~in the bowlers table matches~~ = ti.bowler ID ~~in the embedded SELECT statement~~（依次基于场次 ID 相同将投球手得分表内连接到联赛场次表、基于联赛 ID 内连接到联赛表，再从原始得分超过 180 的行中选择联赛日期、联赛地点、场次 ID、投球手 ID 和原始得分，并将这个结果集命名为 ti；基于投球手 ID 相同将投球手表左外连接到结果集 ti，并选择投球手姓名、联赛日期、联赛地点、场次 ID 和原始得分）

SQL

```
SELECT Bowlers.BowlerLastName || ', ' ||
    Bowlers.BowlerFirstName AS BowlerName,
    TI.TourneyDate, TI.TourneyLocation,
    TI.MatchID, TI.RawScore FROM Bowlers
```

```
LEFT OUTER JOIN
    (SELECT Tournaments.TourneyDate,
        Tournaments.TourneyLocation,
        Bowler_Scores.MatchID,
        Bowler_Scores.BowlerID,
        Bowler_Scores.RawScore
    FROM (Bowler_Scores
     INNER JOIN Tourney_Matches
       ON Bowler_Scores.MatchID =
          Tourney_Matches.MatchID)
     INNER JOIN Tournaments
       ON Tournaments.TourneyID =
          Tourney_Matches.TourneyID
     WHERE Bowler_Scores.RawScore > 180)
   AS TI
ON Bowlers.BowlerID = TI.BowlerID
```

CH09_All_Bowlers_And_Scores_Over_180（106 行）

BowlerName	TourneyDate	TourneyLocation	MatchID	RawScore
Black, Alastair				
Cunningham, David				
Ehrlich, Zachary				
Fournier, Barbara				
Fournier, David				
Hallmark, Alaina				
Hallmark, Bailey				
Hallmark, Elizabeth				
Hallmark, Gary				
Hernandez, Kendra				
Hernandez, Michael				
Kennedy, Angel	2017-11-20	Sports World Lanes	46	185
Kennedy, Angel	2017-10-09	Totem Lanes	22	182
<< 其他行 >>				

- **Recipes 数据库**

❖ **说明**　你猜到了，在这个示例中，也必须先对内连接的结果进行筛选，再将需要从中获取所有行的表外连接到筛选结果。

“列出未在任何菜品中使用的食材。”

转换/整理　Select ingredient name from ~~the~~ ingredients ~~table~~ left outer join~~ed with the~~ recipe ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~ where recipe ID is null（基于食材 ID 相同将食材表左外连接到菜品食材表，并从菜品 ID 为 Null 的行中选择食材名）

SQL
```
SELECT Ingredients.IngredientName
FROM Ingredients
LEFT OUTER JOIN Recipe_Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID
WHERE Recipe_Ingredients.RecipeID IS NULL
```

CH09_Ingredients_Not_Used（20 行）

IngredientName
Halibut
Chicken, Fryer
Bacon
Iceberg Lettuce
Butterhead Lettuce
Scallop
Vinegar
Red Wine
<< 其他行 >>

“我要从 Recipes 数据库中获取所有的菜品类型、所有的菜品名以及各个菜品的食材序号、食材数量和食材度量单位，还有所有的食材名，并依次按菜品类型描述降序、菜品名和食材序列号升序排列。”

转换/整理

Select ~~the~~ recipe class description, recipe title, ingredient name, recipe sequence number, amount, ~~and~~ measurement description from ~~the~~ recipes ~~table~~ left outer join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matches~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table, then~~ inner join~~ed with the~~ measurements ~~table~~ on measurements.measurement amount ID ~~in the measurements table matches~~ = recipe_ingredients.measurement amount ID ~~in the recipe ingredients table, and then~~ full outer join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table, then finally~~ full outer join~~ed with the~~ recipe classes ~~table~~ on recipe_classes.recipe class ID ~~in the recipe classes table matches~~ = recipes.recipe class ID, ~~sorted~~ order by RecipeClassDescription desc~~ending~~, RecipeTitle, ~~and~~ RecipeSeqNo（依次基于菜品 ID 相同将菜品表左外连接到菜品食材表、基于度量单位 ID 相同内连接到度量单位表、基于食材 ID 相同全外连接到食材表、基于菜品类型 ID 相同全外连接到菜品类型表，选择菜品类型描述、菜品名、食材名、食材序列号、食材数量和食材度量单位描述，并按 RecipeClassDescription 降序、RecipeTitle 和 RecipeSeqNo 升序排列）

SQL

```
SELECT Recipe_Classes.RecipeClassDescription,
    Recipes.RecipeTitle,
    Ingredients.IngredientName,
    Recipe_Ingredients.RecipeSeqNo,
    Recipe_Ingredients.Amount,
    Measurements.MeasurementDescription
FROM (((Recipe_Classes
FULL OUTER JOIN Recipes
   ON Recipe_Classes.RecipeClassID =
      Recipes.RecipeClassID)
   LEFT OUTER JOIN Recipe_Ingredients
   ON Recipes.RecipeID =
      Recipe_Ingredients.RecipeID)
   INNER JOIN Measurements
   ON Measurements.MeasureAmountID =
      Recipe_Ingredients.MeasureAmountID)
   FULL OUTER JOIN Ingredients
   ON Ingredients.IngredientID =
      Recipe_Ingredients.IngredientID
ON Recipe_Classes.RecipeClassID =
   Recipes.RecipeClassID
ORDER BY RecipeClassDescription Desc,
RecipeTitle, RecipeSeqNo
```

❖ **说明**　这个问题在讨论全外连接的那一节解决过，这里列举它旨在让你看到实际结果。在我提供的 Microsoft Access 和 MySQL 版示例数据库中，没有使用这种语法的查询，因为这两个数据库系统都不支持全外连接，但提供了合并（UNION）两个外连接查询结果的解决方案，最终的结果是相同的。有关如何使用 UNION，将在下一章介绍。这里显示的是在 Microsoft SQL Server 或 PostgreSQL 中运行这个查询得到的结果。

CH09_All_Recipe_Classes_All_Recipes（109 行）

RecipeClass Description	RecipeTitle	Ingredient Name	Recipe SeqNo	Amount	Measurement Description
Main course	Irish Stew	Beef	1	1	Pound
Main course	Irish Stew	Onion	2	2	Whole
Main course	Irish Stew	Potato	3	4	Whole
Main course	Irish Stew	Carrot	4	6	Whole
Main course	Irish Stew	Water	5	4	Quarts
Main course	Irish Stew	Guinness Beer	6	12	Ounce
Hors d'oeuvres	Salsa Buena	Jalapeno	1	6	Whole
Hors d'oeuvres	Salsa Buena	Tomato	2	2	Whole
<< 其他行 >>					

❖ **说明**　在 PostgreSQL 执行该查询得到的结果中，你将在第 33 行看到菜品类型 Soup 不包含任何菜品（因此也没有任何食材）。在 SQL Server 中执行该查询时，你将在输出末尾看到很多以 Blue Cheese 或 Halibut 打头的食材没有对应的菜品类型和菜品，同时最后一行表示的是菜品类型 Soup。

9.6　小结

本章介绍了外连接。首先给出了外连接的定义，并将其同第 8 章介绍的内连接做了比较。

接下来阐述了如何编写左/右外连接：先从使用两个表的简单示例着手，再转向嵌入 SELECT 语句以及编写使用多个连接的语句。紧接着指出了外连接和 Null 测试的组合与第 7 章介绍的差集（EXCEPT）运算等价；还讨论了编写使用多个外连接的语句时可能遇到的一些麻烦；最后讨论了一个这样的问题：需要使用多个外连接，但仅使用左/右外连接无法解决。

在讨论全外连接的过程中，指出了有时需要结合使用这种连接以及其他内连接和外连接才能得到正确的答案。此外还简要地阐述了全外连接的一个变种——UNION JOIN。

之后阐述了外连接很有用的原因，并列举了可使用外连接来解决的各种问题。随后列举了十多个外连接使用示例（每个示例数据库都有几个），并阐述了为解决这些问题而编写的 SQL 语句背后的逻辑。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

9.7　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就可以。

- **Sales Orders 数据库**

  (1) 列出从未订购过头盔的顾客。

  提示：为解决这个问题，也必须先使用内连接来找出所有包含头盔的订单，再将结果外连接到 Customers 表。

  解决方案见 CH09_Customers_No_Helmets（3 行）。

  (2) 显示没有任何销售代表（员工）的邮政编码与其相同的顾客。

  解决方案见 CH09_Customers_No_Rep_Same_Zip（18 行）。

  (3) 列出所有的商品及包含它的订单的日期。

  解决方案见 CH09_All_Products_Any_Order_Dates（2681 行）。

- **Entertainment Agency 数据库**

  (1) 显示没有签订任何演出合约的经纪人。
  解决方案见 CH09_Agents_No_Contracts（1 行）。

  (2) 列出没有与任何演唱组合签约的顾客。
  解决方案见 CH09_Customers_No_Bookings（2 行）。

  (3) 列出所有的演唱组合及其签订的演出合约。
  解决方案见 CH09_All_Entertainers_And_Any_Engagements（112 行）。

- **School Scheduling 数据库**

  (1) 列出没有学生注册的课程。
  提示：只需要 Student_Classes 表中的 enrolled 行，而不需要 completed 行和 withdrew 行。
  解决方案见 CH09_Classes_No_Students_Enrolled（118 行）。

  (2) 显示没有指定教员的科目。
  解决方案见 CH09_Subjects_No_Faculty（1 行）。

  (3) 列出当前未注册任何课程的学生。
  提示：需要在学生选课表中找出课程状态为 enrolled 的学生，再找出不在这个集合中的学生。
  解决方案见 CH09_Students_Not_Currently_Enrolled（2 行）。

  (4) 显示所有的教员及其讲授的课程。
  解决方案见 CH09_All_Faculty_And_Any_Classes（135 行）。

- **Bowling League 数据库**

  (1) 显示没有比赛数据的场次。
  解决方案见 CH09_Matches_Not_Played_Yet（1 行）。

  (2) 显示所有的联赛及其已举行的比赛场次。
  解决方案见 CH09_All_Tourneys_Match_Results（174 行）。

- **Recipes 数据库**

  (1) 显示不包含任何菜品的菜品类型。
  解决方案见 CH09_Recipe_Classes_No_Recipes（1 行）。

  (2) 显示所有的食材及其被用来制作的菜品。
  解决方案见 CH09_All_Ingredients_Any_Recipes（108 行）。

  (3) 列出菜品类型 Salad（沙拉）、Soup（汤菜）和 Main course（主菜）及其包含的菜品。
  解决方案见 CH09_Salad_Soup_Main_Courses（9 行）。

  (4) 显示所有的菜品类型及其包含的菜品。
  解决方案见 CH09_All_RecipeClasses_And_ Matching_Recipes（16 行）。

第 10 章 UNION

“我祈求虔诚的人能够虔诚地请愿，为这次联合祈祷。”

——得克萨斯州英雄山姆·休斯顿

本章涵盖如下主题：

- 何谓 UNION
- 使用 UNION 编写查询
- UNION 的用途
- 语句举例
- 小结
- 练习

第 7 章介绍了三种基本的集合运算——交集、差集和并集；第 8 章演示了如何使用内连接基于键值将结果集关联起来，从而执行与交集运算等价的操作；第 9 章讨论了如何使用外连接和 Null 测试来执行差集运算。本章将介绍如何执行第三种运算——UNION。

10.1　何谓 UNION

UNION 让你能够从多个类似的结果集中选择行，并将它们合并为单个结果集。请注意，这里说的是行而不是列。在第 8 章和第 9 章，你学习了如何使用连接将多个结果集中的列组合在一起。当你执行连接操作时，来自不同结果集的列将并排出现。例如，当你使用连接来获取 Recipe_Classes 表中的 RecipeClassDescription 和 Recipes 表中的 RecipeTitle 时，结果集将如图 10-1 所示。

RecipeClassDescription	RecipeTitle
Main course	Irish Stew
Main course	Fettuccini Alfredo
Main course	Pollo Picoso
Main course	Roast Beef
Main course	Huachinango Veracruzana (Red Snapper, Veracruz style)
Main course	Tourtière (French-Canadian Pork Pie)
Main course	Salmon Filets in Parchment Paper
Vegetable	Garlic Green Beans
<< 其他行 >>	

图 10-1　使用连接从两个表中获取数据

先大致看一看基本 UNION 的语法，如图 10-2 所示。

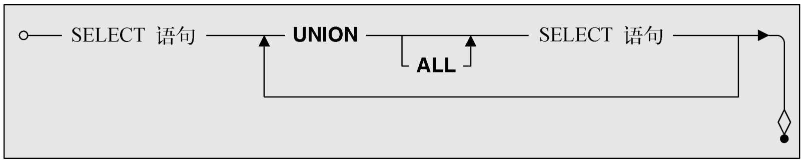


图 10-2　基本 UNION 语句的语法图

UNION 将来自两个结果集的行交织在一起。你通过编写 SELECT 语句来定义各个结果集。在这种 SELECT 语句中，不仅可在 FROM 子句中包含复杂的连接，还可包含 WHERE、HAVING 和 GROUP BY 子句（GROUP BY 和 HAVING 子句将分别在第 13 章和第 14 章介绍）。然后，你使用关键字 UNION 将它们关联起来。如果你从 Recipe_Classes 表获取 RecipeClassDescription，从 Recipes 表获取 RecipeTitle，并使用 UNION 将它们关联起来，结果将如图 10-3 所示。

RecipeClassDescription
Asparagus
Coupe Colonel
Dessert
Fettuccini Alfredo
Garlic Green Beans
Hors d'oeuvres
Huachinango Veracruzana (Red Snapper, Veracruz style)
Irish Stew
<< 其他行 >>

图 10-3　使用 UNION 从两个表获取数据

注意，结果集中只有一列，其名称来自在 SELECT 表达式中指定的第一个表中相应的列，但同时包含来自 RecipeTitle 列的信息（Asparagus）和来自 RecipeClassDescription 列的信息（Dessert）。换而言之，来自这两列的数据不是并排显示，而在垂直方向上交替出现。

如果你仔细研究图 10-2 所示的语法图，可能会问：可选关键字 ALL 到底是做什么用的呢？如果你省略这个关键字，数据库系统将消除值相同的行。例如，如果 RecipeClassDescription 和 RecipeTitle 列都包含 Dessert，最终的结果集中将只有一个 Dessert 行。相反，如果你指定了关键字 ALL，将不会删除重复的行。请注意，包含关键字 ALL 时，UNION 的效率也可高得多，因为在这种情况下，数据库系统将不需要做额外的工作——找出并删除重复的行。如果你确定使用 UNION 合并的查询不包含重复的行（或不在乎包含重复的行），请务必指定关键字 ALL。

要执行 UNION，涉及的两个结果集必须满足特定的要求。首先，在使用 UNION 关联的两条 SELECT 语句中，在关键字 SELECT 后面指定的输出列数必须相同，这样两个结果集的列数才相同；其次，对应的列必须在 SQL 标准看来是可比较的。

❖ **说明**　SQL:2016 完整标准允许 UNION 不相似的集合，但大多数商业实现只支持该标准的基本（入门）部分。在你的数据库系统中，也许能够以更具创意的方式使用 UNION。

第 6 章讨论过，对于字符值，应只将其同字符值进行比较；对于数字值，应只将其同数字值进行比较；对于日期时间值，应只将其同日期时间值进行比较。虽然有些数据库系统允许对不同的数据类型进行比较，但将字符值（如 John）同数字值（如 55）进行比较真的毫无意义。对于两个给定的列，如果在 WHERE 子句中对其进行比较是有意义的，那么这两个列就是可比较的；SQL 标准规定，要 UNION 来自两个不同结果集中的列，它们的数据类型必须是可比较的，说的就是这个意思。

10.2　使用 UNION 编写查询

在前面介绍内连接和外连接的两章中，你学习了如何使用 SELECT、FROM 和 WHERE 子句来编写 SELECT 语句，

其重点是如何在 FROM 子句中编写复杂的连接。为创建 UNION，必须编写 SELECT **表达式**，使用运算符将多条 SELECT 语句关联起来。在其中的每条 SELECT 语句中，都可根据需要使用或简单或复杂的 FROM 子句。

10.2.1　使用简单的 SELECT 语句

我们从简单的着手，先来创建两条简单 SELECT 语句（即其 FROM 子句只使用了一个表）的 UNION。图 10-4 是对两条简单 SELECT 语句执行 UNION 操作的语法图。

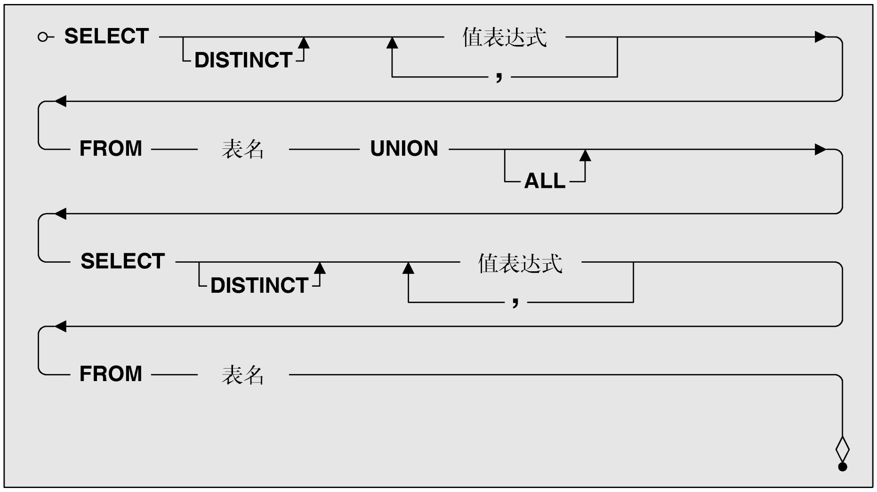


图 10-4　使用 UNION 来组合两条简单的 SELECT 语句

不同于连接，UNION 的所有操作都发生在用来合并 SELECT 语句的 UNION 运算符中。前面说过，如果你省略可选关键字 ALL，数据库系统将消除所有的重复行，这意味着在最终的结果集中，包含的行数可能少于参与 UNION 的各个结果集的行数之和。相反，如果你指定了关键字 ALL，最终结果集中的行数将等于参与 UNION 的各个结果集的行数之和。

> ❖ **说明**　SQL 标准还定义了 CORRESPONDING 子句，你可将其放在关键字 UNION 后面，指出你要通过比较两个结果集中的同名列来执行 UNION。你还可进一步限制要比较哪些列，方法是在关键字 CORRESPONDING 后面指定一系列列名。我发现还没有主要的商业实现支持这项功能，但你使用的数据库系统在未来的版本中可能支持它。

我们来创建一个简单的 UNION：使用示例数据库 Sales Orders 生成一个顾客和供应商邮寄清单。图 10-5 指出了需要用到的两个表。

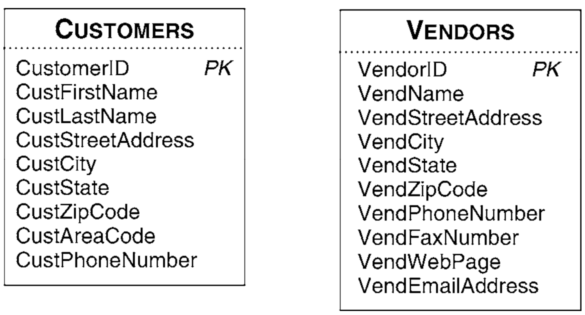


图 10-5　示例数据库 Sales Orders 中的 Customers 表和 Vendors 表

请注意，这两个表之间不存在任何“自然”关系，但它们确实包含含义和数据类型都类似的列。在邮寄清单中，需要姓名（名称）、街道地址、城市、州和邮政编码；由于在这两个表中，这些字段都是可比较的字符数据，因此无须担心数据类型方面的问题（有些数据库设计人员可能将邮政编码列的数据类型指定为数字，但这也没有关系，只要两个表中的邮政编码列的数据类型是可比较的就可以）。

一个问题是，在 Vendors 表中，名称是一列，而在 Customers 表中，有两个姓名列：CustFirstName 和 CustLastName。为了从这两个表中提取相同数量的列，需要编写一个表达式，它使用 Customers 表中的两列来生成一列，以便与 Vendors 表中的单个名称列 UNION。下面来编写查询。

❖ **说明** 贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”方法。

“生成单个邮寄清单，其中包含顾客的姓名、地址、城市、州和邮政编码以及供应商的名称、地址、城市、州和邮政编码。”

<table>
<tr><td>转换</td><td>Select customer full name, customer address, customer city, customer state, and customer ZIP Code from the customers table combined with vendor name, vendor address, vendor city, vendor state, and vendor ZIP Code from the vendors table（从顾客表中选择姓名、地址、城市、州和邮政编码，从供应商表中选择名称、地址、城市、州和邮政编码，然后将这两个结果集合并）</td></tr>
<tr><td>整理</td><td>Select customer full name, customer address, customer city, customer state, ~~and~~ customer ZIP Code from ~~the~~ customers ~~table combined with~~ union Select vendor name, vendor address, vendor city, vendor state, ~~and~~ vendor ZIP Code from ~~the~~ vendors ~~table~~</td></tr>
<tr><td>SQL</td><td><pre>SELECT Customers.CustLastName || ', ' ||
    Customers.CustFirstName AS MailingName,
Customers.CustStreetAddress, Customers.CustCity,
    Customers.CustState, Customers.CustZipCode
FROM Customers
UNION
SELECT Vendors.VendName,
   Vendors.VendStreetAddress, Vendors.VendCity,
   Vendors.VendState, Vendors.VendZipCode
FROM Vendors</pre></td></tr>
</table>

请注意，其中的每条 SELECT 语句都生成 5 列，但我必须使用表达式将 Customers 表中的两个姓名列合并成一列。这两条 SELECT 语句生成的所有列都是字符数据，因此不存在与可比较相关的任何问题。

你可能会问：这个查询输出的列都叫什么名字呢？问得好！SQL 标准规定，当对应的列同名时（如第一条 SELECT 语句生成的第 4 列与第二条 SELECT 语句生成的第 4 列同名），这个名称就是输出列的名称。在列名不同时（就像刚才的示例中那样），SQL 标准是这样规定的：查询表达式包含 UNION 或 INTERSECT，且涉及的表中相应列的名称不同时，结果列的名称取决于实现。

用通俗的语言说，就是由数据库系统决定给输出列指定什么名称。只要数据库系统指定的名称不是参与 UNION 的结果集中其他列的名称，它就符合 SQL 标准。大多数商业数据库系统默认使用第一条 SELECT 语句中指定的列名。对前面的示例来说，这意味着结果列名将为 MailingName、CustStreetAddress、CustCity、CustState 和 CustZipCode。

请注意，我没有在 UNION 中指定关键字 ALL。虽然顾客姓名与供应商名称相同的可能性不大（至于地址、城市、州和邮政编码，就不用考虑了），但我还是想要避免重复的邮寄地址。如果确定要进行合并的多个集合中没有重复的行，可指定关键字 ALL。指定 ALL 通常可提高查询的速度，因为这样数据库系统就无须做额外的工作了——消除重复的行。

10.2.2 合并复杂的 SELECT 语句

可以想见，根据需要完成的任务，使用 UNION 运算符合并的 SELECT 语句可能非常复杂。唯一的限制是，两条 SELECT 语句最终生成的列数相同，且对应列的数据类型是可比较的。

假设你要生成一个清单，其中包含所有的顾客及其订购的自行车以及所有的供应商及其供应的自行车。首先，我们来确定需要用到哪些表。图 10-6 显示了为将顾客与商品关联起来需要用到的表。

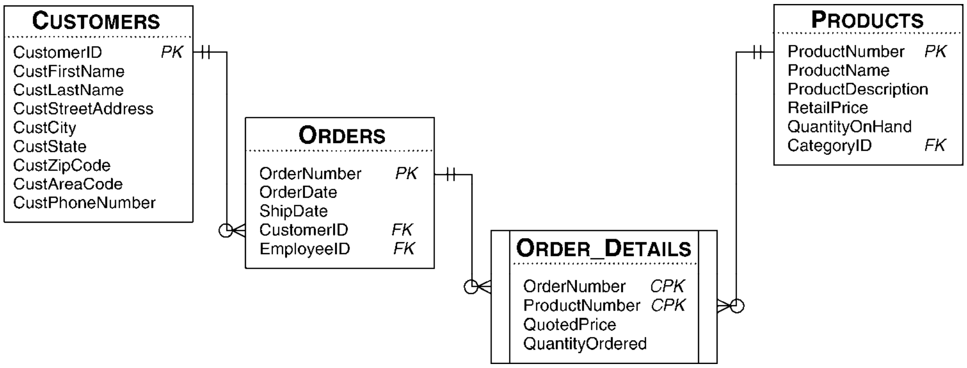


图 10-6　将顾客关联到其订购的商品的表间关系

看起来需要连接 4 个表。要找出供应商及其销售的商品，需要用到如图 10-7 所示的表。

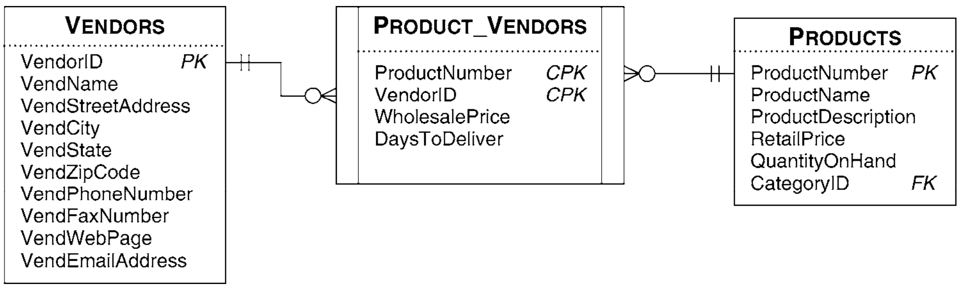


图 10-7　将供应商关联到其销售的商品的表间关系

第 8 章讨论过，可通过嵌套多个 JOIN 子句来连接多个表，以获取解决复杂问题所需的信息。为复习这一点，图 10-8 显示了连接三个表的语法。

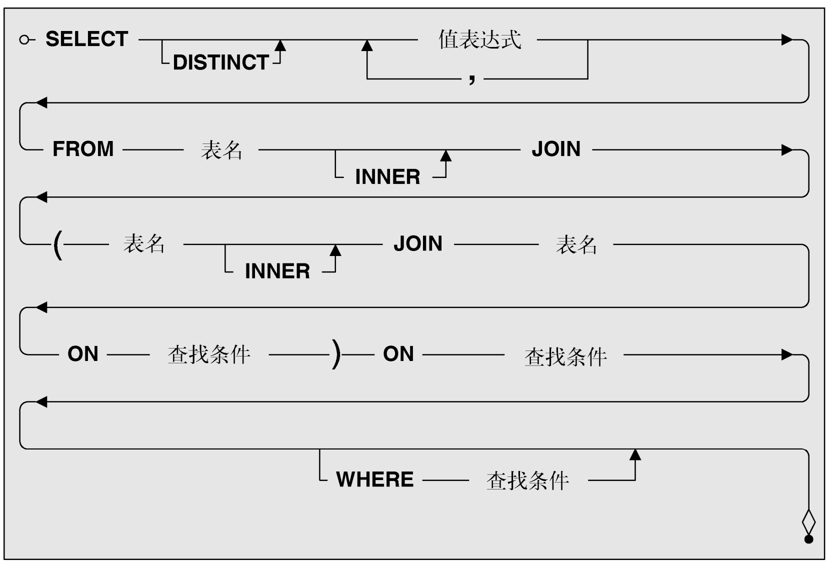


图 10-8　连接三个表的语法

现在万事俱备，可编写一个复合内连接来获取顾客的信息，插入关键字 UNION，再编写一个复合内连接来获取供应商信息。

“列出顾客及其订购的自行车以及供应商及其销售的自行车。”

转换	Select customer full name and product name from the customers table joined with the orders table on customer ID in the customers table matches customer ID in the orders table, then joined with the order details table on order number in the orders table matches order number in the order details table, and then joined with the products table on product number in the products table matches product number in the order details table where product name contains ‘bike’, combined with select vendor name and product name from the vendors table joined with the product vendors table on vendor ID in the vendors table matches vendor ID in the product vendors table, and then joined with the products table on product number in the products table matches product number in the product vendors table where product name contains ‘bike’（依次基于顾客 ID 相同将顾客表内连接到订单表、基于订单号相同内连接到订单详情表、基于商品编号相同内连接到商品表，并从商品名包含 bike 的行中选择顾客姓名和商品名；依次基于供应商 ID 相同将供应商表内连接到商品供应商表、基于商品编号相同内连接到商品表，并从商品名包含 bike 的行中选择供应商名称和商品名；将上述两个结果集合并）
整理	Select customer full name ~~and~~ product name from ~~the~~ customers ~~table~~ join~~ed with the~~ orders ~~table~~ on customers.customer ID ~~in the customers table matches~~ = orders.customer ID ~~in the orders table, then~~ join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table matches~~ = order_details.order number ~~in the order details table, and then~~ join~~ed with the~~ products ~~table~~ on products.product number ~~in the products table matches~~ = order_details.product number ~~in the order details table~~ where product name ~~contains~~ like ‘%bike%’, ~~combined with~~ union select vendor name ~~and~~ product name from ~~the~~ vendors ~~table~~ join~~ed with the~~ product vendors ~~table~~ on vendors.vendor ID ~~in the vendors table matches~~ = product_vendors.vendor ID ~~in the product vendors table, and then~~ join~~ed with the~~ products ~~table~~ on products.product number ~~in the products table matches~~ = product_vendors.product number ~~in the product vendors table~~ where product name ~~contains~~ like ‘%bike%’

SQL

```
SELECT Customers.CustLastName || ', ' ||
    Customers.CustFirstName AS FullName,
    Products.ProductName, 'Customer' AS RowID
FROM ((Customers INNER JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
  ON Orders.OrderNumber =
     Order_Details.OrderNumber)
INNER JOIN Products
  ON Products.ProductNumber =
     Order_Details.ProductNumber
WHERE Products.ProductName LIKE '%bike%'
UNION
SELECT Vendors.VendName, Products.ProductName,
     'Vendor' AS RowID
FROM (Vendors
INNER JOIN Product_Vendors
  ON Vendors.VendorID = Product_Vendors.VendorID)
INNER JOIN Products
  ON Products.ProductNumber =
     Product_Vendors.ProductNumber
WHERE Products.ProductName LIKE '%bike%'
```

这条 SQL 语句很长，但确实解决了问题！注意，在两条 SELECT 语句中，我都添加了一个字符串字面量（我将其命名为 RowID），旨在让你清楚地知道哪些行来自 Customers 表，哪些行来自 Vendors 表。你可能想在第一条 SELECT 语句中插入关键字 DISTINCT，因为大主顾可能多次订购同一种型号的自行车，但由于没有在 UNION 中指定关键字 ALL，因此这个查询将消除所有重复的行。如果你添加关键字 DISTINCT，就可能导致数据库系统做额外的工作——两次试图消除重复的行。

编写使用 UNION 的查询时，建议先编写涉及的各条 SELECT 语句，这样就很容易将它们复制到新的查询中，再在它们之间添加关键字 UNION。

10.2.3 多次使用 UNION

到目前为止，只演示了如何使用一次 UNION 来合并两个结果集。实际上，可在第二条 SELECT 语句后面再添加关键字 UNION 和一条 SELECT 语句。虽然有些实现限制了使用 UNION 可合并的结果集数量，但从理论上说，你可不断添加关键字 UNION 和 SELECT 语句，直到满意为止。

假设你要使用三个不同的表（Customers、Employees 和 Vendors）来生成一个邮寄清单——可能是为了发出节日问候。图 10-9 所示的语法图演示了如何生成这个清单。

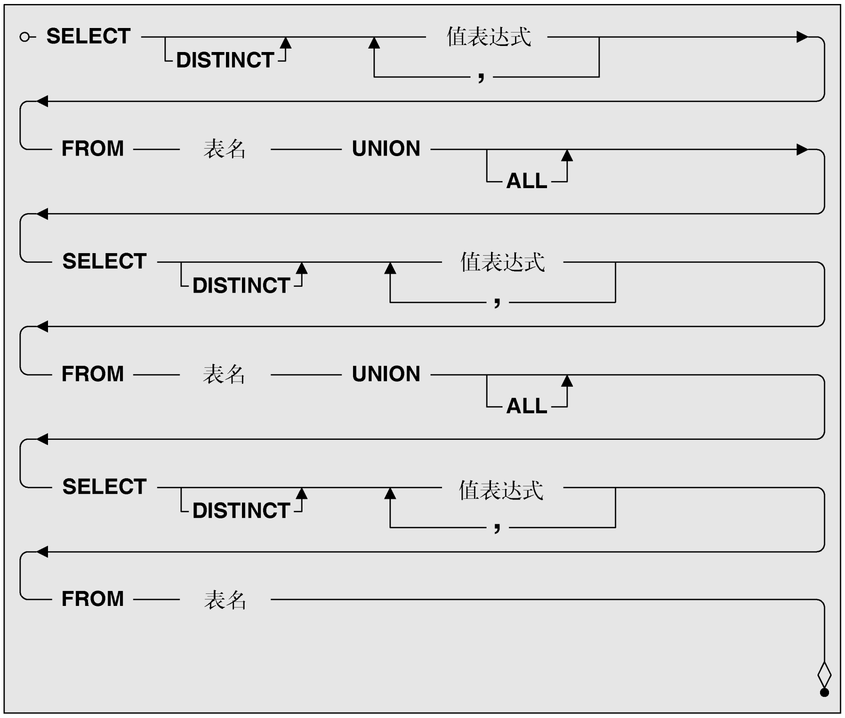


图 10-9　编写三表 UNION

从中可知，需要编写 3 条 SELECT 语句，分别从 Customers、Employees 和 Vendors 表中获取所有的姓名（名称）和地址，再两次使用关键字 UNION 将这些结果集合并（为简化流程，我在这个示例中将转换和整理合二为一了）。

“生成一个包含顾客、员工和供应商的邮寄清单。”

<table>
<tr><td>转换/整理</td><td>Select customer full name, customer street address, customer city, customer state, ~~and~~ customer ZIP Code from ~~the~~ customers ~~table combined with~~ union Select employee full name, employee street address, employee city, employee state, ~~and~~ employee ZIP Code from ~~the~~ employees ~~table combined with~~ union Select vendor name, vendor street address, vendor city, vendor state, ~~and~~ vendor ZIP Code from ~~the~~ vendors ~~table~~（从顾客表中选择姓名、街道地址、城市、州和邮政编码；从员工表中选择姓名、街道地址、城市、州和邮政编码；从供应商表中选择名称、街道地址、城市、州和邮政编码；使用 UNION 将这三个结果集合并）</td></tr>
<tr><td>SQL</td><td>

```
SELECT Customers.CustFirstName || ' ' ||
     Customers.CustLastName AS CustFullName,
     Customers.CustStreetAddress,
     Customers.CustCity,
    Customers.CustState, Customers.CustZipCode
FROM Customers
UNION
SELECT Employees.EmpFirstName || ' ' ||
     Employees.EmpLastName AS EmpFullName,
     Employees.EmpStreetAddress, Employees.EmpCity,
     Employees.EmpState,
     Employees.EmpZipCode
FROM Employees
UNION
SELECT Vendors.VendName, Vendors.VendStreetAddress,
    Vendors.VendCity, Vendors.VendState,
     Vendors.VendZipCode
FROM Vendors
```

</td></tr>
</table>

当然，如果你要筛选出特定的城市、州或邮政编码范围，可在任何或所有 SELECT 语句中添加 WHERE 子句。例如，如果你要创建一个邮寄清单，其中只包含特定州的顾客、员工和供应商，就必须在每个嵌入的 SELECT 语句中添

加 WHERE 子句。你也可只在一条 SELECT 语句中指定筛选器。例如，通过这样做，可生成包含得克萨斯州供应商以及所有顾客和员工的邮寄清单。

10.2.4　对 UNION 的结果集进行排序

如果要对 UNION 的结果进行排序呢？在很多数据库系统中，结果集看起来像是以从左到右的顺序依次按输出列排序的；例如，在前一小节编写的三表 UNION 中，结果集中的行是依次按姓名（名称）、街道地址等排序的。

为了让邮政部门满意（进而可能获得大量邮寄折扣），请按邮政编码对结果集排序。为此，可添加一个 ORDER BY 子句，但诀窍是必须将这条子句放在最后面，即最后一条 SELECT 语句的后面，因为它应用于 UNION 的结果，而不是最后一条 SELECT 语句。图 10-10 演示了具体做法。

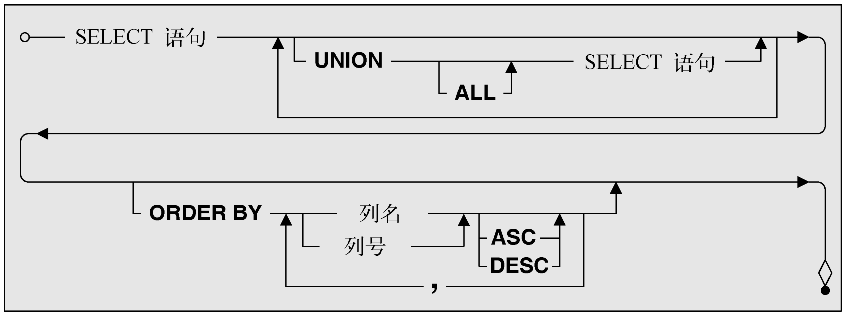


图 10-10　在使用 UNION 的查询中添加排序说明

从该图可知，可反复添加关键字 UNION 和 SELECT 语句任意次，以获取要合并的所有结果集，但 ORDER BY 子句必须放在最后。你可能会问：在 ORDER BY 子句中，该如何指定列名或列号呢？别忘了，你是要对 SELECT 表达式中所有组成部分的结果进行排序，而前面讨论过，输出列名因实现而异，但大部分数据库系统使用第一条 SELECT 语句生成的列名。

你也可指定相对列号——第一个输出列的编号为 1。在输出姓名、街道地址、城市、州和邮政编码的查询中，要按邮政编码排序，需要指定列号 5（邮政编码为第 5 列）。

下面使用这两种方式对邮寄清单进行排序。使用列名指定排序方式的语法如下：

```
SQL          SELECT Customers.CustFirstName || ' ' ||
                 Customers.CustLastName AS CustFullName,
                 Customers.CustStreetAddress, Customers.CustCity,
                 Customers.CustState, Customers.CustZipCode
             FROM Customers
             UNION
             SELECT Employees.EmpFirstName || ' ' ||
                 Employees.EmpLastName AS EmpFullName,
                 Employees.EmpStreetAddress, Employees.EmpCity,
                 Employees.EmpState, Employees.EmpZipCode
             FROM Employees
             UNION
             SELECT Vendors.VendName, Vendors.VendStreetAddress,
                 Vendors.VendCity, Vendors.VendState,
                 Vendors.VendZipCode
             FROM Vendors
             ORDER BY CustZipCode
```

当然，这里假设要用于排序的输出列的名称是第一条 SELECT 语句生成的列名。使用相对列号来指定排序方式的语法类似于下面这样：

```
SQL          SELECT Customers.CustFirstName || ' ' ||
                 Customers.CustLastName AS CustFullName,
                 Customers.CustStreetAddress,
```

```
    Customers.CustCity,
    Customers.CustState, Customers.CustZipCode
FROM Customers
UNION
SELECT Employees.EmpFirstName || ' ' ||
    Employees.EmpLastName AS EmpFullName,
    Employees.EmpStreetAddress, Employees.EmpCity,
    Employees.EmpState, Employees.EmpZipCode
FROM Employees
UNION
SELECT Vendors.VendName, Vendors.VendStreetAddress,
    Vendors.VendCity, Vendors.VendState,
    Vendors.VendZipCode
FROM Vendors
ORDER BY 5
```

10.3　UNION 的用途

你可能不会像使用内连接和外连接那样频繁地使用 UNION，但你很可能使用它来合并多个来自不同表的类似结果集。虽然可使用 UNION 来合并来自同一个或同一组表的结果集，但这种问题通常使用包含复杂 WHERE 子句的简单 SELECT 语句就能解决。10.4 节列举了几个这样的示例，即相比于使用 UNION，使用 WHERE 子句解决问题的效率更高。

下面以示例数据库为例，列举了一些可使用 UNION 来解决的问题。

"显示所有顾客和员工的姓名和地址。"

"列出所有订购了自行车的顾客，以及所有订购了头盔的顾客（这个问题也可使用一条包含复杂 WHERE 子句的 SELECT 语句来解决）。"

"生成一个包含所有顾客和供应商的邮寄清单。"

"列出订购了自行车的顾客以及提供自行车的供应商。"

"生成一个包含经纪人和演唱组合的清单。"

"生成一个包含所有顾客和演唱组合的清单。"

"生成一个清单，其中包含喜欢现代音乐的顾客以及演奏现代音乐的演唱组合。"

"生成一个包含所有学生和教员的邮寄清单。"

"显示艺术课程的平均分不低于 85 的学生以及讲授艺术课且评分不低于 9 的教员。"

"找出在保龄球馆 Thunderbird Lanes 比赛时原始得分不低于 155 的投球手，以及在保龄球馆 Bolero Lanes 比赛时原始得分不低于 155 的投球手（这个问题也可使用一个包含复杂 WHERE 子句的 SELECT 语句来解决）。"

"列出哪些球队（球队的名称和队长）在哪些联赛场次中开始使用的是单数球道，哪些球队（球队的名称和队长）在哪些联赛场次中开始使用的是双数球道。"

"生成一个索引表，其中包含所有的菜品名和食材。"

"列出所有的食材及其默认度量单位，还有各种被用于制作菜品的食材及其在各个菜品中的度量单位。"

10.4　语句举例

至此，你明白了使用 UNION 编写查询的机制，还知道一些可使用 UNION 来解决的问题类型。下面来看一些 UNION 使用示例，这些示例都摘自示例数据库，演示了如何使用 UNION 来合并结果集。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库，查询名以 CH10 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明**　这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。

- **Sales Orders 数据库**

“列出所有顾客和员工的姓名和地址（包括重复的内容），并按邮政编码排序。”

转换/整理　　Select cust~~omer~~ first name, cust~~omer~~ last name, cust~~omer~~ street address, cust~~omer~~ city, cust~~omer~~ state, ~~and~~ cust~~omer~~ ZIP Code from ~~the~~ customers ~~table combined with~~ union all Select emp~~loyee~~ first name, emp~~loyee~~ last name, emp~~loyee~~ street address, emp~~loyee~~ city, emp~~loyee~~ state, ~~and~~ emp~~loyee~~ ZIP Code from ~~the~~ employees ~~table~~, order by ZIP Code（从顾客表中选择名、姓、街道地址、城市、州和邮政编码，从员工表中选择名、姓、街道地址、城市、州和邮政编码，并使用 UNION 合并这两个结果集）

SQL

```
SELECT Customers.CustFirstName,
    Customers.CustLastName,
    Customers.CustStreetAddress,
    Customers.CustCity,
    Customers.CustState, Customers.CustZipCode
FROM Customers
UNION ALL
SELECT Employees.EmpFirstName,
    Employees.EmpLastName,
    Employees.EmpStreetAddress, Employees.EmpCity,
    Employees.EmpState, Employees.EmpZipCode
FROM Employees
ORDER BY CustZipCode
```

CH10_Customers_UNION_ALL_Employees（36 行）

CustFirst Name	CustLast Name	CustStreet Address	CustCity	CustState	CustZip Code
Estella	Pundt	2500 Rosales Lane	Dallas	TX	75260
Robert	Brown	672 Lamont Ave	Houston	TX	77201
Kirk	DeGrasse	455 West Palm Ave	San Antonio	TX	78284
Kirk	DeGrasse	455 West Palm Ave	San Antonio	TX	78284
Angel	Kennedy	667 Red River Road	Austin	TX	78710
Maria	Patterson	3445 Cheyenne Road	El Paso	TX	79915
Mark	Rosales	323 Advocate Lane	El Paso	TX	79915
Caleb	Viescas	4501 Wetland Road	Long Beach	CA	90809
		<< 其他行 >>			

（请注意，Kirk DeGrasse 肯定既是顾客又是员工。）

“列出所有订购了自行车的顾客，还有所有订购了头盔的员工。”

转换/整理　　Select cust~~omer~~ first name, cust~~omer~~ last name, ~~and the constant~~ ‘Bike’ from ~~the~~ customers ~~table~~ join~~ed with the~~ orders ~~table~~ on customers.customer ID ~~in the customers table matches~~ = orders.customer ID ~~in the orders table~~, ~~then~~ join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table matches~~ = order_details.order number ~~in the order details table~~, ~~and then~~ join~~ed with the~~ products ~~table~~ on product number ~~in the products table matches~~ = order_details.product number ~~in the order details table~~ where product name ~~contains~~ like ‘%bike%,’ ~~combined with~~ union Select cust~~omer~~ first name, cust~~omer~~ last name, ~~and the constant~~ ‘Helmet’ from ~~the~~ customers ~~table~~ join~~ed with the~~ orders ~~table~~ on customers.customer ID ~~in the customers table matches~~ = orders.customer ID ~~in the orders table~~, ~~then~~ join~~ed with the~~ order details ~~table~~ on orders. order number ~~in the orders table~~ matches = order_details.order number ~~in the order details table~~, ~~and then~~ join~~ed with the~~ products ~~table~~ on product number ~~in the products table matches~~ = order_details.product number ~~in the order details table~~ where product name ~~contains~~ like ‘%helmet%’（依次基于顾客 ID 相同将顾客表内连接到订单表、基于订单号相同内连接到订单详情表、基于商品编号相同内连接到商品表，从商品名包含 bike 的行中选择顾客的名和姓，并添加一个内容固定为 Bike 的列；依次基于顾客 ID 相同将顾客表内连接到订单表、基于订单号相同内连接到订单详情表、基于商品编号相同内连接到商品表，从商品名包含 helmet 的行中选择顾客的名和姓，并添加一个内容固定为 Helmet 的列；使用 UNION 合并这两个结果集）

```
SQL           SELECT Customers.CustFirstName,
                  Customers.CustLastName, 'Bike' AS ProdType
              FROM ((Customers INNER JOIN Orders
                ON Customers.CustomerID = Orders.CustomerID)
              INNER JOIN Order_Details
                ON Orders.OrderNumber =
                  Order_Details.OrderNumber)
              INNER JOIN Products
                ON Products.ProductNumber =
                  Order_Details.ProductNumber
              WHERE Products.ProductName LIKE '%bike%'
              UNION
              SELECT Customers.CustFirstName,
                   Customers.CustLastName, 'Helmet' AS ProdType
              FROM ((Customers INNER JOIN Orders
                ON Customers.CustomerID = Orders.CustomerID)
              INNER JOIN Order_Details
                ON Orders.OrderNumber = Order_Details.OrderNumber)
              INNER JOIN Products
                ON Products.ProductNumber =
                   Order_Details.ProductNumber
              WHERE Products.ProductName LIKE '%helmet%'
```

CH10_Customer_Order_Bikes_UNION_Customer_Order_Helmets（52 行）

CustFirstName	CustLastName	ProdType
Alaina	Hallmark	Bike
Andrew	Cencini	Bike
Andrew	Cencini	Helmet
Angel	Kennedy	Bike
Angel	Kennedy	Helmet
Caleb	Viescas	Bike
Caleb	Viescas	Helmet
Darren	Gehring	Bike
	<< 其他行 >>	

请注意，这个问题也可使用单条包含复杂 WHERE 子句的 SELECT 语句来解决。使用 UNION 的一个优点是，很容易在结果集中添加人造的“集合标识符”列（这里是 ProdType 列），让你知道顾客来自哪个结果集。然而，大多数数据库系统处理 WHERE 子句的速度比处理 UNION 的速度快得多，即便 WHERE 子句包含复杂的查找条件。下面是使用 WHERE 子句解决这个问题的 SQL 语句，但请注意，这将消除表示订购了自行车和头盔的顾客的重复行，因为在结果集中没有包含 ProdType 列。要包含 ProdType 列，必须在 SELECT 子句中使用 CASE 语句，这将在第 19 章介绍。

```
SQL           SELECT DISTINCT Customers.CustFirstName,
                   Customers.CustLastName
              FROM ((Customers INNER JOIN Orders
                ON Customers.CustomerID = Orders.CustomerID)
              INNER JOIN Order_Details
                ON Orders.OrderNumber = Order_Details.OrderNumber)
              INNER JOIN Products
                ON Products.ProductNumber =
                   Order_Details.ProductNumber
              WHERE Products.ProductName LIKE '%bike%'
                OR Products.ProductName LIKE '%helmet%'
```

CH10_Customers_Bikes_Or_Helmets（27 行）

CustFirstName	CustLastName
Alaina	Hallmark
Andrew	Cencini
Angel	Kennedy
Caleb	Viescas

（续）

CustFirstName	CustLastName
Darren	Gehring
David	Smith
Dean	McCrae
Estella	Pundt
<< 其他行 >>	

❖ **说明** 如你所见，在不使用 UNION 的情况下，要消除重复行，需要使用关键字 DISTINCT。而使用 UNION 时，除非指定了关键字 ALL，否则将自动消除重复行。在使用 UNION 的示例中，也可指定关键字 DISTINCT，但这会让数据库系统去做不必要的工作。

- **Entertainment Agency 数据库**

“生成一个包含经纪人和演唱组合的清单。”

转换/整理

Select agent full name, ~~and the constant~~ ‘Agent’ from ~~the~~ agents ~~table combined with~~ union Select ent~~ertainer~~ stage name, ~~and the constant~~ ‘Entertainer’ from ~~the~~ entertainers ~~table~~（从经纪人表中选择姓名并添加一个内容固定为 Agent 的列；从演唱组合表中选择艺名并添加一个内容固定为 Entertainer 的列；使用 UNION 合并这两个结果集）

SQL

```
SELECT Agents.AgtLastName || ', ' ||
     Agents.AgtFirstName AS Name, 'Agent' AS Type
FROM Agents
UNION
SELECT Entertainers.EntStageName,
     'Entertainer' AS Type
FROM Entertainers
```

CH10_Agents_UNION_Entertainers（22 行）

Name	Type
Bishop, Scott	Agent
Carol Peacock Trio	Entertainer
Caroline Coie Cuartet	Entertainer
Coldwater Cattle Company	Entertainer
Country Feeling	Entertainer
Dumbwit, Daffy	Agent
Jazz Persuasion	Entertainer
Jim Glynn	Entertainer
<< 其他行 >>	

- **School Scheduling 数据库**

“显示艺术课成绩不低于 85 的学生以及讲授艺术课且评分不低于 9 的教员。”

转换/整理

Select stud~~ent~~ first name ~~aliased~~ as FirstName, stud~~ent~~ last name ~~aliased~~ as LastName, ~~and~~ grade ~~aliased~~ as Score from ~~the~~ students ~~table~~ join~~ed with the~~ student schedules ~~table~~ on students.student ID ~~in the students table matches~~ = student_schedules.student ID ~~in the student schedules table, then~~ join~~ed with the~~ student class status ~~table~~ on student_class_status.class status ~~in the student class status table matches~~ = student_schedules.class status ~~in the student schedules table, then~~ join~~ed with the~~ classes ~~table~~ on classes.class ID ~~in the classes table matches~~ = student_schedules.class ID ~~in the student schedules table, and then~~ join~~ed with the~~ subjects ~~table~~ on subjects.subject ID ~~in the subjects table matches~~ = classes.subject ID ~~in the classes table~~ where class status description ~~is~~ = ‘completed’ and grade ~~is greater than or equal to~~ >= 85 and category ID ~~is~~ = ‘ART’ ~~combined with~~ union Select sta~~ff~~ first name, sta~~ff~~ last name, ~~and~~ proficiency rating ~~aliased~~ as Score from ~~the~~ staff ~~table~~ join~~ed with the~~ faculty subjects ~~table~~ on staff.staff ID ~~in the staff table matches~~ = faculty_subjects.staff ID ~~in the faculty subjects table, and then~~ join~~ed with the~~ subjects ~~table~~ on subjects.subject ID ~~in the subjects table matches~~ = faculty_subjects.subject ID ~~in the faculty subjects table~~ where proficiency rating ~~is greater than~~ > 8 and category ID ~~is~~ = ‘ART’[依次基于学生 ID 相同将学生表内连接到学生选课表、

基于课程状态相同内连接到学生课程状态表、基于课程 ID 相同内连接到课程表、基于科目 ID 相同内连接到科目表，从课程状态描述为 completed、成绩低于 85 且类别 ID 为 ART 的行中选择学生的名（将其别名指定为 FirstName）、姓（别名为 LastName）和成绩（别名为 Score）；依次基于教员 ID 相同将教员表内连接到教员科目表、基于科目 ID 相同内连接到科目表、从评分高于 8 且类别 ID 为 ART 的行中选择教员的名、姓和评分；使用 UNION 合并这两个结果集］

SQL

```
SELECT Students.StudFirstName AS FirstName,
     Students.StudLastName AS LastName,
     Student_Schedules.Grade AS Score,
     'Student' AS Type
FROM (((Students INNER JOIN Student_Schedules
  ON Students.StudentID =
     Student_Schedules.StudentID)
INNER JOIN Student_Class_Status
  ON Student_Class_Status.ClassStatus =
     Student_Schedules.ClassStatus)
INNER JOIN Classes
  ON Classes.ClassID = Student_Schedules.ClassID)
INNER JOIN Subjects
  ON Subjects.SubjectID = Classes.SubjectID
WHERE Student_Class_Status.ClassStatusDescription =
     'Completed'
  AND Student_Schedules.Grade >= 85
  AND Subjects.CategoryID = 'ART'
UNION
SELECT Staff.StfFirstName, Staff.StfLastName,
     Faculty_Subjects.ProficiencyRating AS Score,
     'Faculty' AS Type
FROM (Staff INNER JOIN Faculty_Subjects
  ON Staff.StaffID = Faculty_Subjects.StaffID)
INNER JOIN Subjects
  ON Subjects.SubjectID = Faculty_Subjects.SubjectID
WHERE Faculty_Subjects.ProficiencyRating > 8
  AND Subjects.CategoryID = 'ART'
```

CH10_Good_Art_Students_And_Faculty（12 行）

FirstName	LastName	Score	Type
Alaina	Hallmark	10	Faculty
George	Chavez	97.81	Student
John	Kennedy	87.65	Student
Kerry	Patterson	99.83	Student
Liz	Keyser	10	Faculty
Mariya	Sergienko	9	Faculty
Michael	Hernandez	10	Faculty
<<其他行>>			

- **Bowling League 数据库**

“列出哪些球队（球队的名称和队长）在哪些联赛场次中开始使用的是单数球道，哪些球队（球队的名称和队长）在哪些联赛场次中开始使用的是双数球道，并按联赛日期和场次编号排序。”

转换/整理

Select tourney location, tourney date, match ID, team name, captain name ~~and the constant~~ ‘Odd Lane’ from the tournaments ~~table~~ join~~ed with the~~ tourney matches ~~table~~ on tournaments.tourney ID ~~in the tournaments table equals~~ = tourney_matches.tourney ID ~~in the tourney matches table, then~~ join~~ed with the~~ teams ~~table~~ on tourney_matches.odd lane team ID ~~in the tourney matches table equals~~ = teams.team ID ~~in the teams table, and then~~ join~~ed with the~~ bowlers ~~table~~ on teams.captain ID ~~in the teams table equals~~ = bowlers.bowler ID ~~in the bowlers table, combined with~~ union all Select tourney location, tourney date, match ID, team name, captain name ~~and the constant~~ ‘Even Lane’ from ~~the~~ tournaments ~~table~~ join~~ed with the~~ tourney matches ~~table~~ on tournaments.tourney ID ~~in the tournaments table equals~~ = tourney_matches.tourney ID ~~in the tourney matches table, then~~ join~~ed with the~~ teams ~~table~~ on tourney_matches.even lane team ID ~~in the tourney matches table equals~~ = teams.team ID ~~in the teams table, and then~~ join~~ed with the~~ bowlers ~~table~~ on teams.captain ID ~~in the teams table equals~~ = bowlers.bowler ID ~~in the bowlers table~~, order by ~~tourney date~~ 2, ~~and match ID~~ 3［依次基于联赛 ID 相同将联赛表内连接到联赛场次表、基于单数赛道球队 ID 与球队 ID 相同内连接到球队表、基于队长 ID 与投球手 ID 相同内连接到投球手表，选择联赛地点、联赛日期、场次 ID、球队名称、队长姓名，并添加一个内容固定为 Odd Lane 的列；依次基于联赛 ID 相同将联赛表内连接到联赛场

	次表、基于双数赛道球队 ID 与球队 ID 相同内连接到球队表、基于队长 ID 与投球手 ID 相同内连接到投球手表，选择联赛地点、联赛日期、场次 ID、球队名称、队长姓名，并添加一个内容固定为 Even Lane 的列；使用 UNION ALL 合并这两个结果集，并按联赛日期（第 2 列）和场次 ID（第 3 列）排序]

SQL

```
SELECT Tournaments.TourneyLocation,
     Tournaments.TourneyDate,
     Tourney_Matches.MatchID, Teams.TeamName,
     Bowlers.BowlerLastName || ', ' ||
     Bowlers.BowlerFirstName AS Captain,
     'Odd Lane' AS Lane
FROM ((Tournaments INNER JOIN Tourney_Matches
  ON Tournaments.TourneyID =
     Tourney_Matches.TourneyID)
INNER JOIN Teams
  ON Teams.TeamID =
     Tourney_Matches.OddLaneTeamID)
INNER JOIN Bowlers
  ON Bowlers.BowlerID = Teams.CaptainID
UNION ALL
SELECT Tournaments.TourneyLocation,
     Tournaments.TourneyDate,
     Tourney_Matches.MatchID, Teams.TeamName,
     Bowlers.BowlerLastName || ', ' ||
     Bowlers.BowlerFirstName AS Captain,
     'Even Lane' AS Lane
FROM ((Tournaments INNER JOIN Tourney_Matches
  ON Tournaments.TourneyID =
     Tourney_Matches.TourneyID)
INNER JOIN Teams ON Teams.TeamID =
     Tourney_Matches.EvenLaneTeamID)
INNER JOIN Bowlers
  ON Bowlers.BowlerID = Teams.CaptainID
ORDER BY 2, 3
```

请注意，这里的两条 SELECT 语句几乎完全相同，唯一的差别在于，将 Tourney_Matches 连接到 Teams 表时，第一条 SELECT 语句使用的是 OddLaneTeamID，而第二条 SELECT 语句使用的是 EvenLaneTeamID。另外请注意，在最终的解决方案中，我使用了相对列号（第 2 列和第 3 列）而不是列名（TourneyDate 和 MatchID）来指定排序方式。最后，之所以可以使用 UNION ALL，是因为一个球队不可能与自己比赛。

CH10_Bowling_Schedule（114 行）

Tourney Location	Tourney Date	MatchID	TeamName	Captain	Lane
Red Rooster Lanes	2017-09-04	1	Marlins	Fournier, David	Odd Lane
Red Rooster Lanes	2017-09-04	1	Sharks	Patterson, Ann	Even Lane
Red Rooster Lanes	2017-09-04	2	Barracudas	Sheskey, Richard	Even Lane
Red Rooster Lanes	2017-09-04	2	Terrapins	Viescas, Carol	Odd Lane
Red Rooster Lanes	2017-09-04	3	Dolphins	Viescas, Suzanne	Odd Lane
Red Rooster Lanes	2017-09-04	3	Orcas	Thompson, Sarah	Even Lane
Red Rooster Lanes	2017-09-04	4	Manatees	Viescas, Michael	Odd Lane
Red Rooster Lanes	2017-09-04	4	Swordfish	Rosales, Joe	Even Lane
Thunderbird Lanes	2017-09-11	5	Marlins	Fournier, David	Even Lane
Thunderbird Lanes	2017-09-11	5	Terrapins	Viescas, Carol	Odd Lane
<< 其他行 >>					

- **Recipes 数据库**

“生成一个索引表，其中包含所有的菜品类型、菜品名和食材。”

转换/整理	Select recipe class description, ~~and the constant~~ ‘Recipe Class’ from ~~the~~ recipe classes ~~table combined with~~ union Select recipe title, ~~and the constant~~ ‘Recipe’ from ~~the~~ recipes ~~table combined with~~ union Select ingredient name, ~~and the constant~~ ‘Ingredient’ from ~~the~~ ingredients ~~table~~（从菜品类型表中选择菜品类型描述，并添加一个内容固定为 Recipe Class 的列；从菜品表中选择菜品名，并添加一个内容固定为 Recipe 的列；从食材表中选择食材名，并添加一个内容固定为 Ingredient 的列；使用 UNION 合并这三个结果集）

SQL

```
SELECT Recipe_Classes.RecipeClassDescription
     AS IndexName, 'Recipe Class' AS Type
FROM Recipe_Classes
UNION
SELECT Recipes.RecipeTitle, 'Recipe' AS Type FROM
  Recipes
UNION
SELECT Ingredients.IngredientName,
     'Ingredient' AS Type
FROM Ingredients
```

CH10_Classes_Recipes_Ingredients（101 行）

IndexName	Type
Asparagus	Ingredient
Asparagus	Recipe
Bacon	Ingredient
Balsamic vinaigrette dressing	Ingredient
Beef	Ingredient
Beef drippings	Ingredient
Bird’s custard powder	Ingredient
Black olives	Ingredient
<< 其他行 >>	

10.5　小结

本章首先给出了 UNION 的定义，并指出了使用连接关联两个表与使用 UNION 合并两个表的差别。

接下来阐述了如何使用两条都从单个表中获取列的 SELECT 语句来编写简单的 UNION，以及关键字 ALL 的作用，并建议你在确定查询不会生成重复行或不在乎是否有重复行时使用这个关键字。然后介绍了如何合并两条都连接多个表的复杂 SELECT 语句。接下来演示了如何使用 UNION 来合并两个以上的结果集。最后演示了如何对 UNION 的结果集进行排序。

之后阐述了 UNION 很有用的原因，并列举了使用它可解决的各种问题。在 10.4 节，对于每个示例数据库，都列举了一个或两个在其中使用 UNION 的示例，并阐述了示例中编写的 SQL 语句背后的逻辑。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

10.6　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 列出订购了头盔的顾客以及提供头盔的供应商。

  提示：需要 UNION 两个复杂的连接操作的结果集。

  解决方案见 CH10_Customer_Helmets_Vendor_Helmets（91 行）。

- **Entertainment Agency 数据库**

  (1) 列出所有的顾客和演唱组合。

  提示：请编写一个表达式来合并顾客的姓和名，确保两条 SELECT 语句生成的列数相同。

  解决方案见 CH10_Customers_UNION_Entertainers（28 行）。

  (2) 生成一个清单，其中包含喜欢现代音乐的顾客以及演奏现代音乐的演唱组合。

  提示：要解决这个问题，需要 UNION 两个复杂的连接操作。

  解决方案见 CH10_Customers_Entertainers_Contemporary（5 行）。

- **School Scheduling 数据库**

  (1) 生成一个包含学生和教员的邮寄清单，并按邮政编码排序。

  提示：尝试使用相对行号来指定排序方式。

  解决方案见 CH10_Student_Staff_Mailing_List（45 行）。

- **Bowling League 数据库**

  (1) 找出在保龄球馆 Thunderbird Lanes 比赛时得分不低于 165 的投球手以及在保龄球馆 Bolero Lanes 比赛时得分不低于 150 的投球手。

  提示：这个问题也可使用一个包含复杂 WHERE 子句的 SELECT 语句来解决。

  使用 UNION 的解决方案见 CH10_Good_Bowlers_TBird_Bolero_UNION（129 行）；使用 WHERE 的解决方案见 CH10_Good_Bowlers_TBird_Bolero_WHERE（135 行）。

  (2) 你能解释上一题的两种解决方案返回的行数为何不同吗？

  提示：尝试在第一个解决方案中使用 UNION ALL。

- **Recipes 数据库**

  (1) 列出所有的食材及其默认度量单位，还有各种被用于制作菜品的食材及其在各个菜品中的度量单位。

  提示：要解决这个问题，需要使用一个简单的连接和一个复杂的连接。

  解决方案见 CH10_Ingredient_Recipe_Measurements（144 行）。

第 11 章 子查询

“我们不能用提出问题的思维方式解决问题。”

——阿尔伯特·爱因斯坦

本章涵盖如下主题：

- 何谓子查询
- 作为列表达式的子查询
- 作为筛选器的子查询
- 子查询的用途
- 语句举例
- 小结
- 练习

前三章中演示了多种使用多个表中数据的方式。本书前面介绍的所有技巧都致力于将信息子集关联起来，这些信息子集可能是一列或多列、一行或多行，它们是由表或嵌入在 FROM 子句中的查询提供的。前一章还探讨了如何使用 UNION 运算符来合并信息集。本章将探索如何从表或查询中获取单列，并将其用作 SELECT 或 WHERE 子句中的值表达式。

你将通过本章学到两个要点。

- 对于特定的问题，使用 SQL 来解决的方式总是有多种。事实上，本章将演示用于解决前几章列举的问题的新方式。
- 可编写复杂的筛选器，它们不仅仅依赖于你在 FROM 子句中指定的表。这是一个重要的概念，因为需要根据从一个表中筛选出的内容从另一个相关表中获取行时，要获取正确数量的行，唯一的办法是在 WHERE 子句中使用子查询。这将在本章后面更详细地探讨。

> ❖ **说明** 本章将介绍的是高级概念，并假定你已阅读并充分理解了第 7、8、9 章的内容。

11.1 何谓子查询

简单地说，**子查询**就是为编写正确的查询语句而在 SELECT 语句的子句中嵌入的 SELECT 表达式。本章将对子查询做出更正式的定义，并演示如何在除 FROM 子句外的其他子句中使用它。

SQL 标准定义了三类子查询。

- **行子查询**：返回多列但不超过一行的嵌入式 SELECT 表达式。
- **表子查询**：返回一列或多列以及零行或多行的嵌入式 SELECT 表达式。
- **标量子查询**：只返回一列且不超过一行的嵌入式 SELECT 表达式。

11.1.1　行子查询

你已编写过在 FROM 子句中嵌入 SELECT 语句的查询，这种查询让你能够先筛选行，再将结果表连接到其他表或查询。这种查询被称为表子查询，稍后将介绍。

行子查询是一种特殊的 SELECT 语句，返回多列但不超过一行。

在 SQL 标准中，可使用行子查询来创建**行值构造器**（row value constructor）。编写 WHERE 子句时，使用的查找条件通常是将表中的某列与另一列或字面量进行比较，但在 SQL 标准中，可编写这样的查找条件，即将多个值同另一组值进行比较，这两组值被称为逻辑行，也叫行值构造器。要指定一系列要比较的值，可在括号内输入一个值列表，也可使用行子查询从表中获取一行。坏消息是支持这种语法的商业数据库系统不多。

这可能很有用，为什么呢？假设有一个 Products 表，在两列中存储了复合零件标识符。标识符的第一部分可能是字符，指出了零件的类型（SKUClass），如 CPU 或 DSK；标识符的第二部分可能是一个数字，用于在类型中标识零件（SKUNumber）。假设你要找出标识符为 DSK09775 或更大值的零件。下面演示了如何在 WHERE 子句使用行值构造器来解决这个问题：

```
SQL         SELECT SKUClass, SKUNumber, ProductName
            FROM Products
            WHERE
               (SKUClass, SKUNumber)
               >= ('DSK', 9775)
```

上述 WHERE 子句查找 SKUClass 和 SKUNumber 的组合大于 DSK 和 9775 的组合的零件，与下面的 SQL 语句等价：

```
SQL         SELECT SKUClass, SKUNumber, ProductName
            FROM Products
            WHERE (SKUClass > 'DSK')
            OR ((SKUClass = 'DSK')
            AND (SKUNumber >= 9775))
```

对于这里的比较表达式的第二部分，可使用一条返回单行的 SELECT 语句（一个行子查询）来代替（可能需要使用 WHERE 子句将结果限制为一行）。大多数商业数据库系统不支持行值构造器和行子查询，因此本章对它们的介绍就到这里。

11.1.2　表子查询

等等，在前三章中，我是不是演示了如何在 FROM 子句中嵌入返回多行多列的 SELECT 表达式？答案是肯定的。在前几章，我大量地使用了表子查询来获取复杂的结果，再将其嵌入到另一个查询的 FROM 子句中。本章将介绍如何使用表子查询来提供要比较的一系列值，供你在谓词 IN 中使用（这个谓词在第 6 章做了简要介绍）。此外还将介绍一些新的比较谓词，它们只能与表子查询结合使用。

11.1.3　标量子查询

本章还将介绍如何在可使用值表达式的地方使用标量子查询。标量子查询让你能够从一个表中获取单列或计算表达式，而这个表不必是在主查询的 FROM 子句中指定的。对于标量子查询获取的单个值，你可将其用于 SELECT 语句的列列表中，也可在 WHERE 子句中将其用作比较值。

11.2　作为列表达式的子查询

第 5 章详细介绍了如何使用表达式在查询的输出中生成计算列，但没有指出也可使用特殊的 SELECT 语句（子查询）从另一个表中获取数据，即便这个表并没有出现在 FROM 子句中。

11.2.1　语法

我们来复习一点基础知识，看看简单的 SELECT 语句是什么样的，如图 11-1 所示。

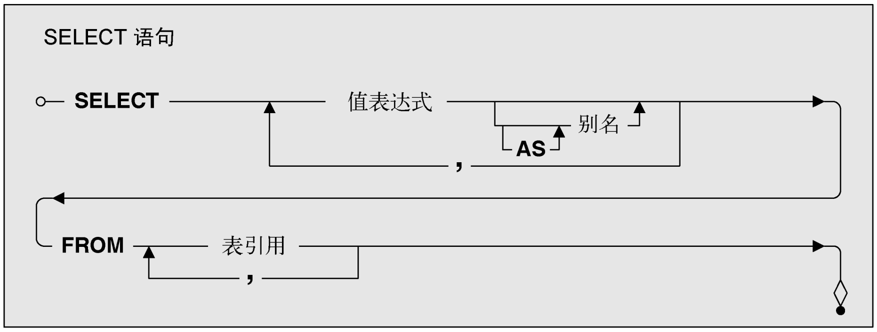
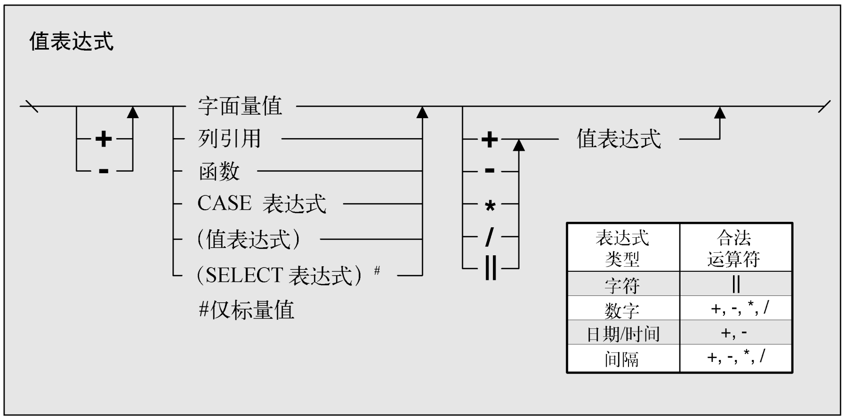


图 11-1　简单 SELECT 语句的语法图

这看似简单，实则一点也不简单。事实上，其中的“值表达式”部分可以非常复杂。图 11-2 显示了可作为值表达式的各种内容。



表达式类型	合法运算符
字符	||
数字	+, -, *, /
日期/时间	+, -
间隔	+, -, *, /

图 11-2　值表达式的语法图

第 5 章介绍了如何使用字面量值、列引用和函数来编写简单的值表达式，CASE 表达式将在第 19 章介绍。注意，列表中也包含 SELECT 表达式。这意味着在关键字 SELECT 后面的表达式列表中，可嵌入标量子查询。本章前面说过，标量子查询是只返回一列且不超过一行的 SELECT 表达式。这合乎逻辑，因为这种子查询用于替换单个列名或结果为单列的表达式。

你可能会问：标量子查询为何很有用呢？它让你能够从一个表或查询中获取单个值，并将其添加到查询的输出中。在子查询的 FROM 子句中指定的数据源（即表或查询）根本不必是在外部查询的 FROM 子句中指定的。在大多数情况下，需要在子查询的 WHERE 子句中指定条件，确保它返回的行数不超过 1。你甚至可以在子查询的查找条件中引用外部查询返回的值，以获取与当前行相关的数据。

下面来看一些简单的示例，它们只使用了示例数据库 Sales Orders 中的 Customers 和 Orders 表。图 11-3 说明了这两个表之间的关系。

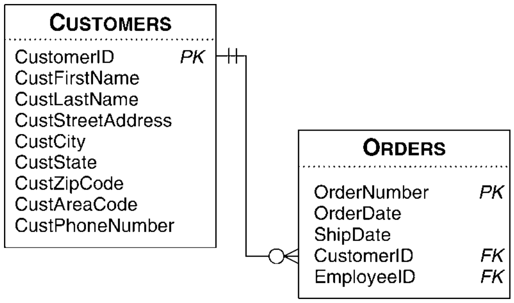


图 11-3　Customers 表和 Orders 表

下面来编写一个查询，列出特定日期的所有订单，并使用子查询从 Customers 表中获取相关顾客姓什么。

> ❖ **说明**　贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”流程。另外，在“整理”步骤中，我还将子查询的各个部分用括号括起，并在 SQL 中尽可能缩进子查询，以帮助你搞明白我是如何使用它们的。

“列出在 2017 年 10 月 3 日发货的所有订单以及与每个订单相关的顾客的姓。”

转换	Select order number, order date, ship date, and also select the related customer last name out of the customers table from the orders table where ship date is October 3, 2017（从订单表中选择发货日期为 2017 年 10 月 3 日的订单的编号、下单日期、发货日期，并从顾客表中选择相关顾客的姓）
整理	Select order number, order date, ship date, ~~and also~~ (select ~~the related~~ custo~~mer~~ last name ~~out of the~~ from customers ~~table~~) from ~~the~~ orders ~~table~~ where ship date ~~is~~ = ~~October 3, 2017~~ ‘2017-10-03’

SQL
```
SELECT Orders.OrderNumber, Orders.OrderDate,
   Orders.ShipDate,
   (SELECT Customers.CustLastName
   FROM Customers
   WHERE Customers.CustomerID =
      Orders.CustomerID)
FROM Orders
WHERE Orders.ShipDate = '2017-10-03'
```

请注意，在子查询中，我将 CustomerID 限制为从 Orders 表中取回的各行的 CustomerID 列的值。如果不这样做，这个子查询将获取 Customers 表中的所有行。别忘了，这个子查询必须是标量子查询，即从一行中返回一个值的查询，因此我必须采取措施，将返回的内容限制为不超过一行。CustomerID 是 Customers 表的主键，因此可以肯定，查找条件 Customers.CustomerID = Orders.CustomerID 会且只会返回一行。

如果你明白了第 8 章介绍的内连接概念，可能会问：为何要像刚才介绍的那样解决这个问题，而不在外部查询的 FROM 子句中将 Orders 表连接到 Customers 表呢？这里的重点是通过简单的示例演示如何使用子查询来生成输出列。实际上，就这个问题而言，你也许确实应该使用如下利用内连接的查询来解决：

SQL
```
SELECT Orders.OrderNumber, Orders.OrderDate,
   Orders.ShipDate, Customers.CustLastName
FROM Customers
INNER JOIN Orders
ON Customers.CustomerID = Orders.OrderID
WHERE Orders.ShipDate = '2017-10-03'
```

11.2.2　聚合函数 COUNT 和 MAX 简介

你明白使用子查询来生成输出列的基本概念后，我们来开阔你的视野，看看为何这种功能很有用。首先，需要概述两个聚合函数（下一章将详细介绍所有的聚合函数）。

SQL 标准定义了很多用于在查询中计算值的函数，其中的一类是聚合函数，让你能够根据结果集中的一系列行计算出单个值。例如，你可使用聚合函数来计算行数，在一系列行中找出最小值或最大值，根据结果集计算平均值或累计值。

下面来介绍这些函数中的两个，然后说说为何它们在子查询中最有用。图 11-4 说明了函数 COUNT 和 MAX 的语法，它们可在 SELECT 子句中生成一个输出列。

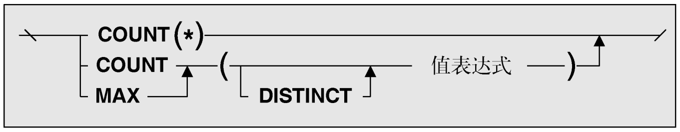


图 11-4　使用聚合函数 COUNT 和 MAX

可使用 COUNT 来确定结果集包含的行数或非 Null 值数量。要确定整个结果集包含多少行，可使用 COUNT(*)。如果你使用 COUNT(column_name)指定了结果集中特定的列，数据库系统将计算该列的值不为 Null 的行数。你还可使用关键字 DISTINCT 计算不同值的个数。

同理，使用 MAX 可找出指定列中的最大值。如果指定的值表达式的数据类型为数字，结果将为最大的数字值；如果指定的值表达式为字符数据类型，最大值将取决于数据库系统使用的排序序列；如果指定的值表达式的数据类型为日期或时间，将返回最后的日期或时间。

下面在子查询中使用这些函数来解决两个有趣的问题。

“列出所有顾客的姓名并计算他们各自下了多少个订单。”

转换	Select customer first name, customer last name, and also select the count of orders from the orders table for this customer from the customers table（从顾客表中选择顾客的名、姓，并从订单表中选择每位顾客下的订单数）
整理	Select cust~~omer~~ first name, cust~~omer~~ last name, ~~and also~~ (select ~~the~~ count ~~of orders~~ (*) from ~~the~~ orders ~~table for this customer~~ where orders.customer ID = customers.customer ID) from ~~the~~ customers ~~table~~
SQL	SELECT Customers.CustFirstName, Customers.CustLastName, (SELECT COUNT(*) FROM Orders WHERE Orders.CustomerID = Customers.CustomerID) AS CountOfOrders FROM Customers

作为输出列的子查询开始变得有趣起来了！第五部分将更详细地介绍如何创造性地使用聚合函数，但如果你只是想计算相关的行数，使用子查询是一种不错的方式。事实上，如果你所需的只是顾客姓名及其下的订单数，这将是解决问题的唯一方式。如果你在主查询的 FROM 子句中添加 Orders 表（`FROM Customers INNER JOIN Orders ON Customers.CustomerID = Orders.CustomerID`），则对于下了多个订单的顾客，将返回多行。第 13 章将介绍另一种根据顾客姓名将行分组的方式。

下面来看一个可使用聚合函数 MAX 来解决的有趣问题。

“列出所有顾客及其最后一次下单的日期。”

转换	Select customer first name, customer last name, and also select the highest order date from the orders table for this customer from the customers table（从顾客表中选择顾客的名和姓，并从订单表中选择每位顾客最后的下单日期）
整理	Select cust~~omer~~ first name, cust~~omer~~ last name, ~~and also~~ (select ~~the highest~~ max(order date) from ~~the~~ orders ~~table for this customer~~ where orders.customer ID = customers.customer ID) from ~~the~~ customers ~~table~~
SQL	SELECT Customers.CustFirstName, Customers.CustLastName, (SELECT MAX(OrderDate) FROM Orders WHERE Orders.CustomerID = Customers.CustomerID) AS LastOrderDate FROM Customers

可以想见，以这种方式使用 MAX 从相关表中查找最大或最近的日期效果很好。11.5 节将介绍使用这些函数的其他方式。

11.3　作为筛选器的子查询

第 6 章介绍了如何通过添加 WHERE 子句来筛选取回的信息，还介绍了如何使用或简单或复杂的比较来获取所需的行。下面演示如何将子查询用作比较参数，进行更复杂的筛选。

11.3.1　语法

再来看一眼图 11-1 所示的 SELECT 语句，看看在 WHERE 子句中使用简单谓词来编写查询的语法。图 11-5 显示了简化的语法图。

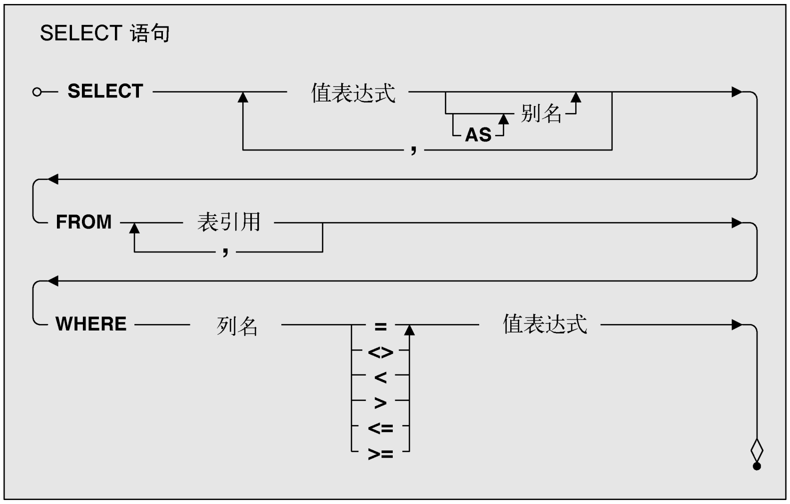


图 11-5　使用简单的比较谓词来筛选结果

图 11-2 指出过，值表达式可以是子查询。在图 11-5 所示的简单示例中，将值表达式同单列进行比较，因此值表达式必须是单个值，即返回一列且不超过一行的标量子查询。下面来解决一个简单的问题，它要求对子查询返回的值进行比较。在这个示例中，我要获取每位顾客的最后一个订单的详情。图 11-6 显示了需要用到的表。

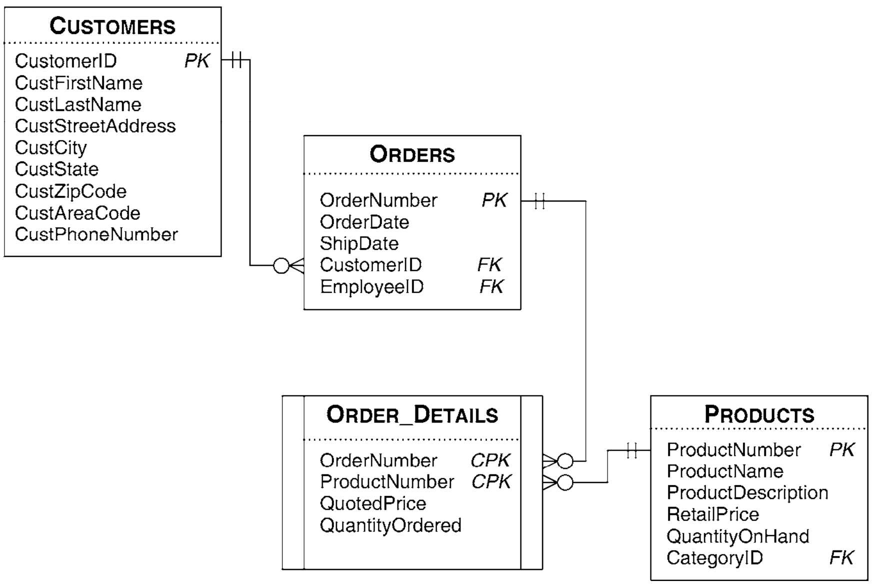


图 11-6　为列出订单详情需要用到的表

“列出所有顾客以及每位顾客最后一个订单的详情。”

<table>
<tr><td>转换</td><td>Select customer first name, customer last name, order number, order date, product number, product name, and quantity ordered from the customers table joined with the orders table on customer ID in the customers table equals customer ID in the orders table, then joined with the order details table on order number in the orders table equals order number in the order details table, and then joined with the products table on product number in the products table equals product number in the order details table where the order date equals the maximum order date from the orders table for this customer（依次基于顾客 ID 相同将顾客表内连接到订单表、基于订单号相同内连接到订单详情表、基于商品编号相同内连接到商品表；基于顾客 ID 相同从订单表中选择顾客的订单，使用 MAX 函数找出顾客最后一个订单的下单日期；在前面的结果集中，从下单日期与刚才计算得到的日期相同的行中选择顾客的名和姓、订单号、下单日期、商品编号、商品名和订购的数量）</td></tr>
<tr><td>整理</td><td>Select cust~~omer~~ first name, cust~~omer~~ last name, order number, order date, product number, product name, ~~and~~ quantity ordered from ~~the~~ customers ~~table~~ inner join~~ed with the~~ orders ~~table~~ on customers.customer ID ~~in the customers table equals~~ = orders.customer ID ~~in the orders table, then~~ inner join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table equals~~ = order_details.order number ~~in the order details table, and then~~ inner join~~ed with the~~ products ~~table~~ on products.product number ~~in the products table equals~~ = order_details.product number ~~in the order details table~~ where ~~the~~ order date ~~equals~~ = (select ~~the~~ max~~imum~~ (order date) from ~~the~~ orders ~~table for this customer~~ where orders.customer ID = customers.customer ID)</td></tr>
<tr><td>SQL</td><td><pre>
SELECT Customers.CustFirstName,
   Customers.CustLastName, Orders.OrderNumber,
   Orders.OrderDate,
   Order_Details.ProductNumber,
   Products.ProductName,
   Order_Details.QuantityOrdered
FROM ((Customers
INNER JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
  ON Orders OrderNumber =
     Order_Details.OrderNumber)
INNER JOIN Products
  ON Products.ProductNumber =
   Order_Details.ProductNumber
WHERE Orders.OrderDate =
   (SELECT MAX(OrderDate)
   FROM Orders AS O2
   WHERE O2.CustomerID = Customers.CustomerID)
</pre></td></tr>
</table>

你可能注意到了，我在第二次引用（即在子查询中引用）Orders 表时，给它指定了一个别名。即便不指定别名，很多数据库系统也知道你指的是子查询中的 Orders 表。事实上，SQL 标准规定，对于任何未限定的引用，应以最内面的查询为起点进行解析。虽然如此，我还是添加了别名，以清楚地指出在子查询的 WHERE 子句中引用的 Orders 表副本是在子查询的 FROM 子句中指定的那个。如果你采用这种做法，查询将容易理解得多，无论是对几个月后再看的你来说，还是对其他人来说都如此。

11.3.2 可与子查询一起使用的特殊谓词关键字

SQL 标准定义了很多特殊的谓词关键字，供你在 WHERE 子句中与子查询一起使用。

1. 集合成员资格谓词：IN

第 6 章介绍了如何在 WHERE 子句中使用关键字 IN 来将列或表达式与值列表进行比较。你现在知道，IN 列表中的每个值表达式都可以是标量子查询。能否使用子查询来生成整个 IN 列表呢？当然能，如图 11-7 所示。

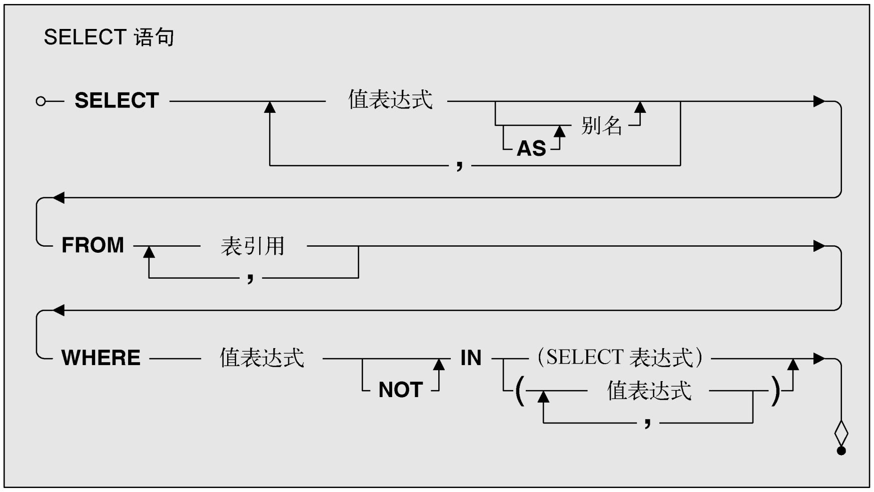


图 11-7 结合使用子查询和 IN 谓词

在这里，可使用一个表子查询，它返回一列，但行数取决于 IN 列表的要求。下面来看一个基于示例数据库 Recipes 的示例。图 11-8 显示了需要用到的表。

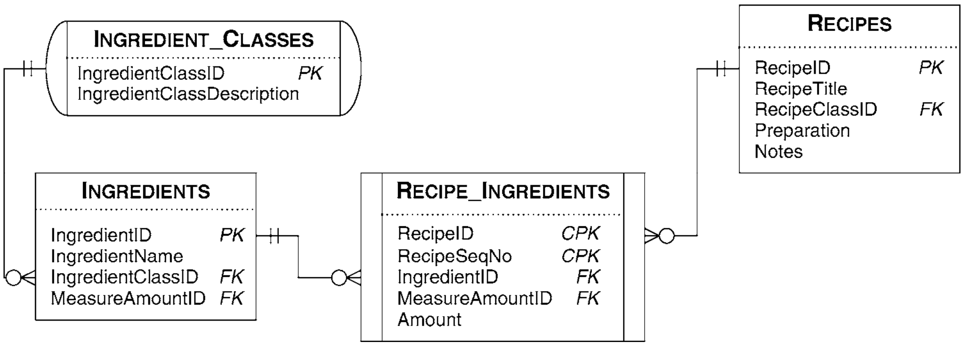


图 11-8 为列出菜品及其使用的食材需要用到的表

假设有位顾客光顾餐厅，他对海鲜钟爱有加。虽然你知道有些菜品使用了海鲜，但不知道数据库中都包含哪些海鲜。然而，你知道有一个 IngredientClassDescription 列，其取值可以为 Seafood，因此可将所有的表都连接起来，并根据 IngredientClassDescription 进行筛选，也可以更有创意，使用子查询和谓词 IN。

“列出所有包含海鲜的菜品。”

转换	Select recipe title from the recipes table where the recipe ID is in the selection of recipe IDs from the recipe ingredients table where the ingredient ID is in the selection of ingredient IDs from the ingredients table joined with the ingredient classes table on ingredient class ID in the ingredients table matches ingredient class ID in the ingredient classes table where ingredient class description is ‘seafood’（基于食材类型 ID 相同将食材类型表内连接到食材表，从食材类型描述为 seafood 的行中选择食材 ID；从菜品食材表中选择食材 ID 位于刚才生成的食材 ID 列表中的行中选择菜品 ID；从菜品表中选择菜品 ID 位于刚才生成的菜品 ID 表中的行中选择菜品名）
整理	Select recipe title from ~~the~~ recipes ~~table~~ where ~~the~~ recipe ID ~~is~~ in ~~the~~ (selecti~~on of~~ recipe IDs from ~~the~~ recipe ingredients ~~table~~ where ~~the~~ ingredient ID ~~is~~ in ~~the~~ (selecti~~on of~~ ingredient IDs from ~~the~~ ingredients ~~table~~ inner join~~ed with the~~ ingredient classes ~~table~~ on ingredients.ingredient class ID ~~in the ingredients table matches~~ = ingredient_classes.ingredient class ID ~~in the ingredient classes table~~ where ingredient class description ~~is~~ = ‘seafood’))

```
SQL        SELECT RecipeTitle
           FROM Recipes
           WHERE Recipes.RecipeID IN
              (SELECT RecipeID
              FROM Recipe_Ingredients
              WHERE Recipe_Ingredients.IngredientID IN
              (SELECT IngredientID
              FROM Ingredients
              INNER JOIN Ingredient_Classes
                ON Ingredients.IngredientClassID =
                   Ingredient_Classes.IngredientClassID
              WHERE
              Ingredient_Classes.IngredientClassDescription
                = 'Seafood'))
```

这是否让你想起了在子查询中可嵌入其他子查询？实际上，可增加一层子查询嵌套，这样就不用在第二个子查询中使用内连接了。对于第二个子查询，可以这样编写：

```
SQL        (SELECT IngredientID
           FROM Ingredients
           WHERE Ingredients.IngredientClassID IN
              (SELECT IngredientClassID
              FROM Ingredient_Classes
              WHERE
              Ingredient_Classes.IngredientClassDescription
                = 'Seafood'))
```

然而，这有点过犹不及了，因为在 IN 子句中嵌入 IN 子句只会让查询更难理解。这里这样做只是为了告诉你可以这样做，但需要重申的是，可以做并不意味着该做！在子查询中使用单个 IN 谓词和较复杂的连接时，语句更容易理解，相信你也这么认为。下面是另一个采取这种做法的解决方案：

```
SQL        SELECT RecipeTitle
           FROM Recipes
           WHERE Recipes.RecipeID IN
              (SELECT RecipeID
              FROM (Recipe_Ingredients
              INNER JOIN Ingredients
                ON Recipe_Ingredients.IngredientID =
                   Ingredients.IngredientID)
              INNER JOIN Ingredient_Classes
                ON Ingredients.IngredientClassID =
                   Ingredient_Classes.IngredientClassID
              WHERE
              Ingredient_Classes.IngredientClassDescription
                = 'Seafood'
```

此时你可能会问：为何要这么麻烦？为何不在外部查询中只使用复杂的连接来解决这个问题呢？原因是这样将得不到正确的答案！实际上，这也将返回 Recipes 表中所有包含海鲜的菜品，但有些菜品可能出现多次。下面尝试在不使用子查询的情况下来解决这个问题，看看为何会返回重复的行。

```
SQL        SELECT RecipeTitle
           FROM ((Recipes
           INNER JOIN Recipe_Ingredients
             ON Recipes.RecipeID =
                Recipe_Ingredients.RecipeID)
           INNER JOIN Ingredients
             ON Recipe_Ingredients.IngredientID =
                Ingredients.IngredientID)
           INNER JOIN Ingredient_Classes
             ON Ingredients.IngredientClassID =
                Ingredient_Classes.IngredientClassID
           WHERE
           Ingredient_Classes.IngredientClassDescription
              = 'Seafood'
```

如果你回过头去看看图 11-8，就知道 Recipes 表中的一行可能对应于 Recipe_Ingredients 表中很多行。在上述 FROM 子句定义的结果集中，包含的行数不少于 Recipe_Ingredients 包含的行数，且有些 RecipeTitle 列值会重复出现很多次。即便添加了筛选器，将食材限定为海鲜（Seafood），对于使用了多种海鲜食材的菜品，在返回的结果集中也会有多行。

当然，你可使用关键字 DISTINCT，但这很可能加重数据库系统的负担——需要做消除重复行的工作。虽然在大多数数据库系统中，将这条语句保存为视图时，它可能是不可更新的，但别忘了，使用 DISTINCT 的视图都是不可更新的，因为 DISTINCT 掩盖了底层行的身份，导致数据库系统不知道该更新哪一行。

如果需要列出除食材名外的其他内容，这种子查询使用技巧将变得非常重要。例如，假设你还要列出使用了海鲜的菜品使用的所有食材。如果你像刚才我那样，在外部查询中使用复杂的连接，并基于食材类型 Seafood 进行筛选，将只能获取海鲜类食材，而无法获取菜品使用的其他食材。再来看一个稍微复杂些的问题。

“列出所有使用了海鲜类食材的菜品以及它们各自使用的所有食材。”

转换	Select recipe title and ingredient name from the recipes table joined with the recipe ingredients table on recipe ID in the recipes table equals recipe ID in the recipe ingredients table, and then joined with the ingredients table on ingredient ID in the ingredients table equals ingredient ID in the recipe ingredients table where the recipe ID is in the selection of recipe IDs from the recipe ingredients table joined with the ingredients table on ingredient ID in the recipe ingredients table equals ingredient ID in the ingredients table, and then joined with the ingredient classes table on ingredient class ID in the ingredients table equals ingredient class ID in the ingredient classes table where ingredient class description is ‘seafood’（依次基于食材 ID 相同将菜品食材表内连接到食材表、基于食材类型 ID 相同内连接到食材类型表，从食材类型描述为 seafood 的行中选择菜品 ID，生成一个菜品 ID 列表；依次基于菜品 ID 相同将菜品表内连接到菜品食材表、基于食材 ID 相同内连接到食材表，从菜品 ID 位于上述菜品 ID 列表中的行中选择菜品名和食材名）
整理	Select recipe title, ~~and~~ ingredient name from ~~the~~ recipes ~~table~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table equals~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table, and then~~ inner join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table equals~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~ where ~~the~~ recipe ID ~~is~~ in ~~the~~ (select~~ion of~~ recipe ID~~s~~ from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on recipe_ingredients.ingredient ID ~~in the recipe ingredients table equals~~ = ingredients.ingredient ID ~~in the ingredients table, and then~~ inner join~~ed with the~~ ingredient classes ~~table~~ on ingredients.ingredient class ID ~~in the ingredients table equals~~ = ingredient_classes.ingredient class ID ~~in the ingredient classes table~~ where ingredient class description ~~is~~ = ‘seafood’)

SQL

```
SELECT Recipes.RecipeTitle,
   Ingredients.IngredientName
FROM (Recipes
INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID
WHERE Recipes.RecipeID IN
   (SELECT RecipeID
   FROM (Recipe_Ingredients
   INNER JOIN Ingredients
     ON Recipe_Ingredients.IngredientID =
        Ingredients.IngredientID)
   INNER JOIN Ingredient_Classes
     ON Ingredients.IngredientClassID =
     Ingredient_Classes.IngredientClassID
WHERE
Ingredient_Classes.IngredientClassDescription
   = 'Seafood')
```

这里的关键之处在于，主查询中复杂的内连接获取选定菜品使用的所有食材，而复杂的子查询返回所有使用了海鲜的菜品的 ID。看起来好像执行了复杂的连接两次，但这样做是有道理的！

2. 限定谓词：ALL、SOME 和 ANY

如你刚才所见，谓词 IN 让你能够将一个列或表达式与一个列表进行比较，看看它是否包含在列表中，即它是否与列表的某个成员相等。如果要确定列或表达式是否大于或小于列表中的任何、所有或某些元素，可使用**限定谓词**。图 11-9 说明了相关的语法。

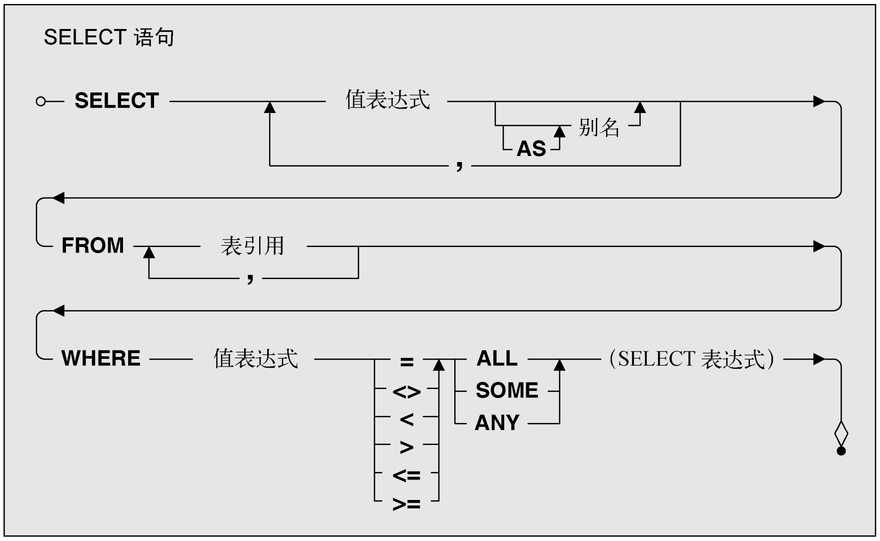


图 11-9　在 SELECT 语句中使用限定谓词

在这里，SELECT 表达式必须是表查询，返回一列和零行或更多行。子查询返回多行时，这些行中的值就组成了一个列表。如你所见，这种谓词由比较运算符和关键字组成，告诉数据库系统如何将运算符应用于列表成员。如果关键字为 ALL，必须对子查询返回的所有值来说比较结果都为真；如果关键字为 SOME 或 ANY，则只要求对列表中的一个值来说比较结果为真。

只要想一想就知道，如果子查询返回多行，= ALL 的结果必将为假，除非子查询返回的值都相同，且左边的值表达式与它们相等。同理，你可能想到了，只要左边的值表达式等于列表中的任何一个值，<> ANY 必然为假。事实上，SQL 标准将 SOME 和 ANY 视为一码事。使用<> SOME 或<> ANY 时，如果左边的值表达式至少与列表中的一个值不相等，结果就为真。另一个令人困惑的地方是，如果子表达式没有返回任何行，那么包含关键字 ALL 的所有比较谓词的结果都为真，而包含关键字 SOME 或 ANY 的所有比较谓词都为假。

下面通过两个示例看看如何实际使用限定谓词。首先来解决一个与数据库 Recipes 相关的问题。请参阅图 11-8 了解要用到哪些表。

"列出使用了牛肉或大蒜的菜品。"

转换	Select recipe title from the recipes table where recipe ID is in the selection of recipe IDs from the recipe ingredients table where ingredient ID equals any of the selection of ingredient IDs from the ingredients table where ingredient name is 'beef' or 'garlic'（在食材表中，从食材名为 beef 或 garlic 的行中选择食材 ID，生成一个食材 ID 列表；在菜品食材表中，从食材 ID 与上述食材 ID 列表中任何一个元素相等的行中选择菜品 ID，生成一个菜品 ID 列表；在菜品表中，从菜品 ID 位于上述菜品 ID 列表中的行中选择菜品名称）
整理	Select recipe title from ~~the~~ recipes ~~table~~ where recipe ID ~~is~~ in ~~the~~ (select~~ion of~~ recipe ID~~s~~ from ~~the~~ recipe ingredients ~~table~~ where ingredient ID ~~equals~~ = any ~~of the~~ (select~~ion of~~ ingredient ID~~s~~ from ~~the~~ ingredients ~~table~~ where ingredient name ~~is~~ in 'beef' ~~or~~ 'garlic'))

SQL	`SELECT Recipes.RecipeTitle`
	`FROM Recipes`
	`WHERE Recipes.RecipeID IN`
	`   (SELECT Recipe_Ingredients.RecipeID`
	`   FROM Recipe_Ingredients`
	`   WHERE Recipe_Ingredients.IngredientID = ANY`
	`   (SELECT Ingredients.IngredientID`
	`   FROM Ingredients`
	`   WHERE Ingredients.IngredientName`
	`   IN ('Beef', 'Garlic')))`

我原本可以使用 IN 而不是 = ANY，你意识到这一点了吗？如果你这么想，那么你完全正确！我原本也可在第一个子查询中将 Recipe_Ingredients 内连接到 Ingredients，以返回所需的 RecipeID 列表。本章开头说过，几乎对于任何

问题来说，使用 SQL 的解决方式都有多种。在有些情况下，使用限定谓词可能会让解决方案更清晰。

下面来解决一个更复杂的问题，以展示限定谓词的真正威力。这个示例使用示例数据库 Sales Orders。需要用到的表如图 11-10 所示。

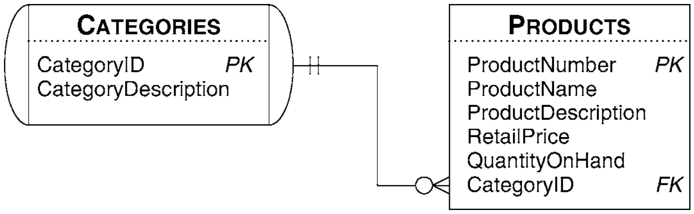


图 11-10　Categories 表和 Products 表之间的关系

"找出价格比任何衣服都高的所有配饰。"

转换	Select product name and retail price from the products table joined with the categories table on category ID in the products table matches category ID in the categories table where category description is 'accessories' and retail price is greater than all the selection of retail price from the products table joined with the categories table on category ID in the products table matches category ID in the categories table where category name is 'clothing'（基于类别 ID 相同将商品表内连接到类别表，从类别名为 clothing 的行中选择零售价，生成一个零售价列表；基于类别 ID 相同将商品表内连接到类别表，从类别描述为 accessories 且零售价高于上述零售价列表中所有元素的行中选择商品名和零售价）
整理	Select product name ~~and~~ retail price from ~~the~~ products ~~table~~ inner join~~ed with the~~ categories ~~table~~ on products.category ID ~~in the products table matches~~ = categories.category ID ~~in the categories table~~ where category description ~~is~~ = 'accessories' and retail price ~~is greater than~~ > all ~~the~~ (select~~ion of~~ retail price from ~~the~~ products ~~table~~ inner join~~ed with the~~ categories ~~table~~ on products.category ID ~~in the products table matches~~ = categories.category ID ~~in the categories table~~ where category name ~~is~~ = 'clothing')

SQL

```
SELECT Products.ProductName,
   Products.RetailPrice
FROM Products
INNER JOIN Categories
  ON Products.CategoryID
     = Categories.CategoryID
WHERE Categories.CategoryDescription =
   'Accessories'
AND Products.RetailPrice > ALL
   (SELECT Products.RetailPrice
   FROM Products
   INNER JOIN Categories
     ON Products.CategoryID =
        Categories.CategoryID
   WHERE Categories.CategoryDescription =
      'Clothing')
```

其中的子查询获取所有衣服的价格，而外部查询列出价格比任何衣服都高的配饰。请注意，在外部查询中，也可以检查零售价是否比子查询取回的最高价格还高，但这里的重点是演示 ALL 的用法。

3. 存在谓词：EXISTS

集合成员资格谓词（IN）和限定谓词（SOME、ANY、ALL）都比较一个值表达式——通常是在外部查询的 FROM 子句中指定的数据源中的一列。在有些情况下，只需知道子查询返回的结果集中是否存在相关的行。第 8 章介绍了一种使用复杂内连接解决 AND 问题的方法，但这种问题也可以使用 EXISTS 来解决。再来看一看第 8 章解决的一个问题。

"找出所有订购了自行车的顾客。"

转换	Select customer ID, customer first name, and customer last name from the customers table where there exists some row from the orders table joined with the order details table on order ID in the orders table equals order ID in the order details table, and then joined with the products table on product ID in the products table equals product ID in the order details table where category ID equals 2（Bikes） and the orders table customer ID equals the customers table customer ID［依次基于订单 ID 相同将订单表内连接到订单详情表、基于商品 ID 相同内连接到商品表，选择类型 ID 为 2（商品为自行车）的行；在顾客表中，从在前述结果集中有相关行（即顾客 ID 相同）的行中选择顾客 ID 以及顾客的名和姓］

<table>
<tr><td>整理</td><td>Select customer ID, cust~~omer~~ first name, ~~and~~ cust~~omer~~ last name from ~~the~~ customers ~~table~~ where ~~there~~ exists ~~some row~~ (select * from ~~the~~ orders ~~table~~ inner join~~ed with the~~ order details ~~table~~ on orders.order ID ~~in the orders table equals~~ = order_details.order ID ~~in the order details table, and then~~ inner join~~ed with the~~ products ~~table~~ on products.product ID ~~in the products table equals~~ = order_details.product ID ~~in the order details table~~ where product name category ID ~~equals~~ = 2 ~~(Bikes)~~ and ~~the~~ orders ~~table~~ customer ID ~~equals~~ = ~~the~~ customers ~~table~~ customer ID)</td></tr>
<tr><td>SQL</td><td><pre>SELECT Customers.CustomerID,
   Customers.CustFirstName,
   Customers.CustLastName
FROM Customers
WHERE EXISTS
   (SELECT *
   FROM (Orders
   INNER JOIN Order_Details
     ON Orders.OrderNumber =
        Order_Details.OrderNumber)
   INNER JOIN Products
     ON Products.ProductNumber =
        Order_Details.ProductNumber
   WHERE Products.CategoryID = 2
     AND Orders.CustomerID =
       Customers.CustomerID)</pre></td></tr>
</table>

请注意，在子查询的 SELECT 子句中，可指定要取回 FROM 子句中指定的任何表中的任何列。我选择了使用表示所有列的简写“*”（11.2.2 节说过，在函数 COUNT 中，使用*或列名没有任何差别，这里也是如此）。换句话说，这个查询要求获取出现在了自行车订单详情中的顾客。由于我没有根据 OrderID 列进行匹配，因此不在乎子查询返回的是哪列。另外，我使用了 `CategoryID = 2` 来走捷径，因为我知道这类商品包含所有的自行车。相比于再连接到 Products 表并检查 ProductName 列是否包含 Bike，这种做法的效率更高。

❖ **说明**　考虑到这个查询很有趣，我使用名称 CH11_Customer_Ordered_Bikes_EXISTS 将其保存到了示例数据库中。有关使用内连接的解决方案，请参阅 CH11_Customers_Ordered_Bikes_JOIN。由于内连接依赖于使用 DISTINCT 来消除重复行，因此这种解决方案是不可更新的。也可使用 IN 来解决这个问题，但我将此作为练习留给你去完成（提示：为方便你检查自己所做的工作，我在示例数据库中存储了一个使用 IN 的示例查询）。

11.4　子查询的用途

至此，你应深入理解了如何使用子查询来生成输出列以及在 WHERE 子句中执行复杂的比较。子查询的用途非常广泛，为了让你明白这一点，最佳的方式是列举一些可使用子查询来解决的问题，并提供一系列子查询使用示例（参见 11.5 节）。

11.4.1　将子查询用作列表达式

本章前面说过，从相关表中取回单个值时，使用子查询的效率可能比使用连接更高。然而，使用子查询取回聚合函数的结果要有趣得多。下一章将更深入地探讨聚合函数。下面是一些可通过使用子查询生成输出列来解决的问题。

“列出所有的供应商及其向我们提供的商品数量。”
“显示所有的商品及其最后一次被订购的日期。”
“显示所有的演唱组合及其签订的演出合约数量。”
“显示所有的顾客及其最后一次下订单的日期。”
“列出所有的教员及其讲授的课程数量。”
“显示所有的科目及其包含的在周一上课的课程数量。”
“显示所有的投球手及其参加的比赛局数。”
“显示所有的投球手及其最高得分。”
“列出所有的肉食食材及其被用来制作的菜品数量。”
“显示所有的菜品类型及其包含的菜品数量。

11.4.2 将子查询用作筛选器

熟悉子查询后，便可扩展用于解决复杂问题的工具包了。本章探讨了很多在 WHERE 子句中将子查询用作筛选器的有趣方式；第 14 章将介绍如何在 HAVING 子句中将子查询用作信息组筛选器。

下面列出了一些可通过在 WHERE 子句中将子查询用作行筛选器来解决的问题。请注意，这些问题中的很多都在本书前面解决过，但在这里，你必须考虑以另一种方式（使用子查询）来解决它们。

> ❖ **说明** 作为提示，我在问题后面的括号内列出了可用来解决当前问题的关键字。

“列出了订购了自行车的顾客（IN）。”
“显示订购了衣服或配饰的顾客（= SOME）。”
“找出所有订购了自行车头盔的顾客（IN）。”
“哪些商品从未被订购过（NOT IN）？”
“列出预订了乡村音乐或乡村摇滚演奏演唱组合的顾客（IN）。”
“找出给顾客 Bonnicksen 或 Rosales 演出过的演唱组合（= SOME）。”
“显示没有签订任何演出合约的经纪人（NOT IN）。”
“列出给顾客 Bonnicksen 演出过的演唱组合（EXISTS）。”
“显示注册了在周二上课的课程的学生（IN）。”
“显示从未退过课的学生（NOT IN）。”
“列出有课程在周三上课的科目（IN）。”
“显示平均得分比其球队中其他所有球员都高的队长（> ALL）。”
“列出平均得分比其球队中其他所有球员都低的投球手（< ALL）。”
“显示包含胡萝卜的菜品使用的所有食材（IN）。”
“列出在某个菜品中使用了且在该菜品中的度量单位不是默认度量单位的食材（<> SOME）。”

11.5 语句举例

至此，你知道了使用子查询来编写查询的机制，还见识了一些可使用子查询来解决的问题类型。下面来看一些示例，它们都使用了一个或多个子查询。这些示例都摘自示例数据库，演示了如何使用子查询来生成输出列或将子查询用作筛选器。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库，查询名以 CH11 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明** 别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录 B。这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。

11.5.1 表达式中的子查询

- **Sales Orders 数据库**

“列出所有的供应商及其向我们提供的商品数量。”

转换/整理　　Select vend~~or~~ name ~~and also~~ (select ~~the~~ count(*) ~~of products~~ from ~~the~~ product vendors ~~table~~ where ~~the~~ product vendor ~~table~~ vendor ID ~~equals~~ = ~~the~~ vendors ~~table~~ vendor ID) from ~~the~~ vendors ~~table~~（从供应商表中选择供应商名称；在商品供应商表中，选择供应商 ID 与当前供应商的 ID 相同的行，并使用 COUNT(*)计算总行数，再将其作为一个新列添加到前面的结果集中）

SQL

```
SELECT VendName,
   (SELECT COUNT(*)
   FROM Product_Vendors
   WHERE Product_Vendors.VendorID =
      Vendors.VendorID)
  AS VendProductCount
FROM Vendors
```

> ❖ **说明**　我给 SELECT 子句中的子查询指定了别名，以便在输出中显示有意义的列名。如果你不这样做，数据库系统将生成类似于 Expr1 这样的列名。

CH11_Vendors_Product_Count（10 行）

VendName	VendProductCount
Shinoman, Incorporated	3
Viscount	6
Nikoma of America	5
ProFormance	3
Kona, Incorporated	1
Big Sky Mountain Bikes	22
Dog Ear	9
Sun Sports Suppliers	5
Lone Star Bike Supply	30
Armadillo Brand	6

- **Entertainment Agency 数据库**

“显示所有的顾客及其签订的最后一个演出合约的演出日期。”

转换/整理　　Select cust~~omer~~ first name, cust~~omer~~ last name, ~~and also~~ (select ~~the highest~~ MAX(start date) from ~~the~~ engagements ~~table~~ where ~~the~~ engagements ~~table~~ customer ID ~~equals~~ = ~~the~~ customers ~~table~~ customer ID) from ~~the~~ customers ~~table~~（从顾客表中选择顾客的名和姓；在演出合约表中，选择顾客 ID 与当前顾客的 ID 相同的行，并使用 MAX(开始日期)找出最后面的演出开始日期，再将其作为一个新列添加到前面的结果集中）

SQL

```
SELECT Customers.CustFirstName,
   Customers.CustLastName,
   (SELECT MAX(StartDate)
   FROM Engagements
   WHERE Engagements.CustomerID =
      Customers.CustomerID)
  AS LastBooking
FROM Customers
```

CH11_Customers_Last_Booking（15 行）

CustFirstName	CustLastName	LastBooking
Doris	Hartwig	2018-02-24
Deb	Waldal	2018-02-18
Peter	Brehm	2018-02-27
Dean	McCrae	2018-02-25
Elizabeth	Hallmark	2018-02-20
Matt	Berg	2018-02-24

（续）

CustFirstName	CustLastName	LastBooking
Liz	Keyser	2018-02-20
Darren	Gehring	
Sarah	Thompson	2018-02-25
	<< 其他行 >>	

❖ **说明**　对有些顾客来说，LastBooking 是空的（Null），这是因为这些顾客没有预订演出。

- **School Scheduling 数据库**

“显示所有的科目及其包含的在周一上课的课程的数量。”

转换/整理　Select subject name ~~and also~~ (select ~~the~~ count(*) ~~of classes~~ from ~~the~~ classes ~~table~~ where Monday schedule ~~is~~ = true and ~~the~~ classes ~~table~~ subject ID ~~equals~~ = ~~the~~ subjects ~~table~~ subject ID) from ~~the~~ subjects ~~table~~（从科目表中选择科目名；在课程表中选择科目 ID 与当前科目的 ID 相同且 MondaySchedule 为真的行，并使用 COUNT(*)计算得到的总函数，再将其作为一个新列添加到前面的结果集中）

SQL

```
SELECT Subjects.SubjectName,
   (SELECT COUNT(*)
   FROM Classes
   WHERE MondaySchedule = 1
   AND Classes.SubjectID = Subjects.SubjectID)
  AS MondayCount
FROM Subjects
```

❖ **说明**　务必使用你所使用的数据库系统支持的真值测试。别忘了，有些数据库系统要求你与关键字 TRUE 或者整数值 1 或–1 比较。

CH11_Subjects_Monday_Count（56 行）

SubjectName	MondayCount
Financial Accounting Fundamentals I	2
Financial Accounting Fundamentals II	0
Fundamentals of Managerial Accounting	0
Intermediate Accounting	0
Business Tax Accounting	0
Introduction to Business	0
Developing A Feasibility Plan	0
Introduction to Entrepreneurship	0
<< 其他行 >>	

❖ **说明**　结果集中没有行时，聚合函数 COUNT 返回 0 而不是 Null。

- **Bowling League 数据库**

“显示所有的投球手及其最高得分。”

转换/整理　Select bowler first name, bowler last name, ~~and also~~ (select ~~the highest~~ MAX(raw score) from ~~the~~ bowler scores ~~table~~ where ~~the~~ bowler scores ~~table~~ bowler ID ~~equals~~ = ~~the~~ bowlers ~~table~~ bowler ID) from ~~the~~ bowlers ~~table~~（从投球手表中选择投球手的名和姓；在投球手得分表中，选择投球手 ID 与当前投球手的 ID 相同的行，并使用 MAX(raw score)计算出最高得分，再将其作为一个新列添加到前面的结果集中）

SQL	`SELECT Bowlers.BowlerFirstName, Bowlers.BowlerLastName, (SELECT MAX(RawScore) FROM Bowler_Scores WHERE Bowler_Scores.BowlerID = Bowlers.BowlerID) AS HighScore FROM Bowlers`

```
SELECT Bowlers.BowlerFirstName,
   Bowlers.BowlerLastName,
   (SELECT MAX(RawScore)
   FROM Bowler_Scores
   WHERE Bowler_Scores.BowlerID =
   Bowlers.BowlerID)
  AS HighScore
FROM Bowlers
```

CH11_Bowler_High_Score（32 行）

BowlerFirstName	BowlerLastName	HighScore
Barbara	Fournier	164
David	Fournier	178
John	Kennedy	191
Sara	Sheskey	149
Ann	Patterson	165
Neil	Patterson	179
David	Viescas	195
Stephanie	Viescas	150
	<< 其他行 >>	

- **Recipes 数据库**

"列出所有的肉食食材及其被用来制作的菜品的数量。"

转换/整理　Select ingredient class description, ingredient name, ~~and also~~ (select ~~the~~ count(*) ~~of rows~~ from ~~the~~ recipe ingredients ~~table~~ where ~~the~~ recipe ingredients ~~table~~ ingredient ID ~~equals~~ = ~~the~~ ingredients ~~table~~ ingredient ID) from ~~the~~ ingredient classes ~~table~~ inner join~~ed~~ ~~with the~~ ingredients ~~table~~ on ingredient_classes.ingredient class ID ~~in the ingredients classes table matches~~ = ingredients.ingredient class ID ~~in the ingredients table~~ where ingredient class description ~~is~~ = 'meat'（基于食材类型 ID 相同将食材类型表内连接到食材表，并从食材类型描述为 meat 的行中选择食材类型描述和食材名；在菜品食材表中，选择食材 ID 与当前食材的 ID 相同的行，并使用 COUNT(*)计算总行数，再将其作为一个新列添加到前面的结果集中）

SQL

```
SELECT Ingredient_Classes.IngredientClassDescription,
   Ingredients.IngredientName,
   (SELECT COUNT(*)
   FROM Recipe_Ingredients
   WHERE Recipe_Ingredients.IngredientID =
       Ingredients.IngredientID)
AS RecipeCount
FROM Ingredient_Classes
INNER JOIN Ingredients
  ON Ingredient_Classes.IngredientClassID =
     Ingredients.IngredientClassID
WHERE
   Ingredient_Classes.IngredientClassDescription
   = 'Meat'
```

CH11_Meat_Ingredient_Recipe_Count（11 行）

IngredientClassDescription	IngredientName	RecipeCount
Meat	Beef	2
Meat	Chicken, Fryer	0
Meat	Bacon	0
Meat	Chicken, Pre-cut	0
Meat	T-bone Steak	0
Meat	Chicken Breast	0
Meat	Chicken Leg	1
Meat	Chicken Wing	0
Meat	Chicken Thigh	1
Meat	New York Steak	0
Meat	Ground Pork	1

11.5.2 筛选器中的子查询

- **Sales Orders 数据库**

"显示订购了衣服或配饰的顾客。"

转换/整理

Select customer ID, cust~~omer~~ first name, cust~~omer~~ last name from ~~the~~ customers ~~table~~ where customer ID ~~is equal to~~ = any ~~of the~~ (selec~~tion~~ of customer ID from ~~the~~ orders ~~table~~ inner join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table matches~~ = order_details.order number ~~in the order details table, then~~ inner join~~ed with the~~ products ~~table~~ on products.product number ~~in the products table matches~~ = order_details.product number ~~in the order details table, and then~~ inner join~~ed with the~~ categories ~~table~~ on categories.category ID ~~in the categories table matches~~ = products.category ID ~~in the products table~~ where category description ~~is~~ = 'clothing' or category description ~~is~~ = 'accessories')（依次基于订单号相同将订单表内连接到订单详情表、基于商品编号相同内连接到商品表、基于类别 ID 相同内连接到类别表，从类别描述为 clothing 或 accessories 的行中选择顾客 ID，生成一个顾客 ID 列表；在顾客表中，从顾客 ID 与前述顾客 ID 列表中任何元素相等的行中选择顾客 ID 以及顾客的名和姓）

SQL

```
SELECT Customers.CustomerID,
   Customers.CustFirstName,
   Customers.CustLastName
FROM Customers
WHERE Customers.CustomerID = ANY
   (SELECT Orders.CustomerID
   FROM ((Orders
   INNER JOIN Order_Details
     ON Orders.OrderNumber =
        Order_Details.OrderNumber)
   INNER JOIN Products
     ON Products.ProductNumber =
        Order_Details.ProductNumber)
   INNER JOIN Categories
     ON Categories.CategoryID =
        Products.CategoryID
   WHERE Categories.CategoryDescription
      = 'Clothing'
     OR Categories.CategoryDescription
        = 'Accessories')
```

CH11_Customers_Clothing_OR_Accessories（27 行）

CustomerID	CustFirstName	CustLastName
1001	Suzanne	Viescas
1002	William	Thompson
1003	Gary	Hallmark
1004	Robert	Brown
1005	Dean	McCrae
1006	John	Viescas
1007	Mariya	Sergienko
1008	Neil	Patterson
	<< 其他行 >>	

❖ **说明** 只是为了好玩，我在解决这个问题时使用了=ANY。你能想出使用 IN 或 EXISTS 的解决方案吗？在示例数据库中，这些解决方案被保存为 CH11_Customers_Clothing_OR_Accessories_IN 和 CH11_Customers_Clothing_OR_Accessories_EXISTS。如果你查看我为 MySQL 提供的这些脚本，将发现我在其中使用的是 ANY，但如果你查看示例数据库，将发现 MySQL 和 PostgreSQL 对实际存储的视图做了转换，在其中使用了 IN。难以置信吧！

- **Entertainment Agency 数据库**

"列出给顾客 Berg 演出过的演唱组合。"

转换/整理

Select entertainer ID, ~~and~~ ent~~ertainer~~ stage name from ~~the~~ entertainers ~~table~~ where ~~there~~ exists (select * ~~some row~~ from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on customers.customer ID ~~in the customers table matches~~ = engagements.customer ID ~~in the engagements table~~ where customer last name ~~is~~ = 'Berg' and ~~the~~ engagements ~~table~~ entertainer ID ~~equals~~ = ~~the~~ entertainers ~~table~~ entertainer ID)［基于顾客 ID 将顾客表内连接到演出合约表，并选择顾客姓 Berg 的行；在演唱组合表中，从在前面的结果集中有相关行（即演唱组合 ID 相同）的行中选择演唱组合 ID 和艺名］

SQL

```
SELECT Entertainers.EntertainerID,
   Entertainers.EntStageName FROM Entertainers WHERE
     EXISTS
   (SELECT *
   FROM Customers
   INNER JOIN Engagements
     ON Customers.CustomerID =
        Engagements.CustomerID
   WHERE Customers.CustLastName = 'Berg'
   AND Engagements.EntertainerID =
     Entertainers.EntertainerID)
```

CH11_Entertainers_Berg_EXISTS（6 行）

EntertainerID	EntStageName
1001	Carol Peacock Trio
1003	JV & the Deep Six
1004	Jim Glynn
1006	Modern Dance
1007	Coldwater Cattle Company
1008	Country Feeling

❖ **说明**　为稍微提高点难度，我决定使用 EXISTS 来解决这个问题。你能使用 IN 来解决吗？有关使用 IN 的解决方案，请参阅 CH11_Entertainers_Berg_IN。

● **School Scheduling 数据库**

"显示从未退过课的学生。"

转换/整理

Select student ID, stud~~ent~~ first name, ~~and~~ stud~~ent~~ last name from ~~the~~ students ~~table~~ where ~~the~~ student ID ~~is~~ not in ~~the~~ (selec~~tion of~~ student ID from ~~the~~ student schedules ~~table~~ inner join~~ed with the~~ student class status ~~table~~ on student_schedules.class status ~~in the student schedules table matches~~ = student_class_status.class status ~~in the student class status table~~ where class status description ~~is~~ = 'withdrew')（基于课程状态相同将学生选课表内连接到学生课程状态表，从课程状态描述为 withdrew 的行中选择学生 ID，生成一个列表；在学生表中，从学生 ID 不在上述列表中的行中选择学生 ID 以及学生的名和姓）

SQL

```
SELECT Students.StudentID,
   Students.StudFirstName,
   Students.StudLastName
FROM Students
WHERE Students.StudentID NOT IN
   (SELECT Student_Schedules.StudentID
   FROM Student_Schedules
   INNER JOIN Student_Class_Status
     ON Student_Schedules.ClassStatus =
        Student_Class_Status.ClassStatus
   WHERE
     Student_Class_Status.ClassStatusDescription
      = 'Withdrew')
```

❖ **说明**　这个查询非常简单，它在子查询中找出退过课的学生，再使用 NOT IN 获取不在这个列表中的学生。你能想出使用外连接的解决方案吗？

CH11_Students_Never_Withdrawn（16 行）

StudentID	StudFirstName	StudLastName
1001	Kerry	Patterson
1003	Betsey	Stadick
1004	Janice	Galvin
1005	Doris	Hartwig
1006	Scott	Bishop
1007	Elizabeth	Hallmark
1008	Sara	Sheskey
1009	Karen	Smith
	<< 其他行 >>	

- **Bowling League 数据库**

“显示让分得分比其球队中其他球员都高的队长。”

转换/整理

Select team name, bowler ID, bowler first name, bowler last name, ~~and~~ handicap score from ~~the~~ bowlers ~~table~~ inner join~~ed with the~~ teams ~~table~~ on bowlers.bowler ID ~~in the bowlers table matches~~ = teams.captain ID ~~in the teams table~~ inner join~~ed with the~~ bowler scores ~~table~~ on bowlers.bowler ID ~~in the bowlers table matches~~ = bowler_scores.bowler ID ~~in the bowler scores table~~ where ~~the~~ handicap score ~~is greater than~~ > all ~~the~~ (select~~ion of~~ handicap score from bowlers as B2 inner join~~ed with the~~ bowler scores ~~table~~ as BS2 on B2.bowler ID ~~in the B2 table matches~~ = BS2.bowler ID ~~in the BS2 table~~ where ~~the~~ B2 ~~table~~ bowler ID ~~is not equal~~ <> ~~the~~ bowlers ~~table~~ bowler ID and ~~the~~ B2 ~~table~~ team ID ~~is equal~~ = ~~to the~~ bowlers ~~table~~ team ID)［依次基于投球手 ID 和队长 ID 相同将投球手表内连接到球队表、基于投球手 ID 相同内连接到投球手得分表；对于该结果集中的每个投球手，都做如下处理：基于投球手 ID 相同将投球手表（别名为 B2）内连接到投球手得分表（别名为 BS2），从投球手 ID 与当前投球手的 ID 不同且球队 ID 与当前投球手的球队 ID 相同的行中，选择让分得分，生成一个让分得分列表；使用> ALL 判断当前投球手的让分得分是否比前述让分得分列表中的所有元素都高，如果是，就从当前投球手对应的行中选择球队名以及投球手的 ID、名和姓］

SQL

```
SELECT Teams.TeamName, Bowlers.BowlerID,
   Bowlers.BowlerFirstName,
   Bowlers.BowlerLastName,
   Bowler_Scores.HandiCapScore
FROM (Bowlers
INNER JOIN Teams
  ON Bowlers.BowlerID = Teams.CaptainID)
INNER JOIN Bowler_Scores
  ON Bowlers.BowlerID = Bowler_Scores.BowlerID
WHERE Bowler_Scores.HandiCapScore > All
   (SELECT BS2.HandiCapScore
   FROM Bowlers AS B2
   INNER JOIN Bowler_Scores AS BS2
      ON B2.BowlerID = BS2.BowlerID
   WHERE B2.BowlerID <> Bowlers.BowlerID
      AND B2.TeamID = Bowlers.TeamID)
```

❖ **说明**　我给在子查询中使用的 Bowlers 和 Bowler_Scores 表的第二个副本指定了别名，让这条 SQL 语句清晰易懂。我不想将当前投球手同他自己比较得分，因为这将导致谓词> ALL 的结果为假；另外，我只想将当前投球手同其球队中的其他球员比较。

CH11_Team_Captains_High_Score（1 行）

TeamName	BowlerID	BowlerFirstName	BowlerLastName	HandiCapScore
Huckleberrys	7	David	Viescas	224

- **Recipes 数据库**

  “显示包含胡萝卜的菜品使用的所有食材。”

> ❖ **说明**　我在第 8 章承诺过，将演示如何使用子查询来解决这个问题。我说到做到！

转换/整理

Select recipe title ~~and~~ ingredient name from ~~the~~ recipes ~~table~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on recipes.recipe ID ~~in the recipes table matches~~ = recipe_ingredients.recipe ID ~~in the recipe ingredients table~~, ~~and then~~ inner join~~ed with the~~ ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~ where recipe ID ~~is~~ in ~~the~~ (select~~ion of~~ recipe ID from ~~the~~ ingredients ~~table~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~ where ingredient name ~~is~~ = ‘carrot’)（基于食材 ID 相同将食材表内连接到菜品食材表，从食材名为 carrot 的行中选择菜品 ID，生成一个菜品 ID 列表；依次基于菜品 ID 相同将菜品表内连接到菜品食材表、基于食材 ID 相同内连接到食材表，从菜品 ID 包含在前述菜品 ID 列表中的行中选择菜品名和食材名）

SQL

```
SELECT Recipes.RecipeTitle,
   Ingredients.IngredientName
FROM (Recipes
INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID
WHERE Recipes.RecipeID
IN
   (SELECT Recipe_Ingredients.RecipeID
   FROM Ingredients
   INNER JOIN Recipe_Ingredients
     ON Ingredients.IngredientID =
        Recipe_Ingredients.IngredientID
   WHERE Ingredients.IngredientName = 'carrot')
```

> ❖ **说明**　如果将 carrot 筛选器放在外部查询中，输出中将只包含胡萝卜这种食材。这个问题要求显示包含胡萝卜的菜品使用的所有食材，因此使用子查询是一种不错的解决方式。这个查询的结果看起来像是按菜品名排序的，虽然没有使用 ORDER BY 子句。如果你要确保在任何数据库系统中都按这样的顺序排列，务必使用 ORDER BY 子句。

CH11_Recipes_Ingredients_With_Carrots（16 行）

RecipeTitle	IngredientName
Irish Stew	Beef
Irish Stew	Onion
Irish Stew	Potato
Irish Stew	Carrot
Irish Stew	Water
Irish Stew	Guinness Beer
Salmon Filets in Parchment Paper	Salmon
Salmon Filets in Parchment Paper	Carrot
Salmon Filets in Parchment Paper	Leek
<< 其他行 >>	

11.6　小结

本章首先给出了 SQL 标准定义的三种子查询（行子查询、表子查询和标量子查询）的定义，并指出本书前面已经介绍过如何在 FROM 子句中使用表查询，还简要地描述了行子查询的用途，并指出了支持它的商业实现不多。

接下来介绍了如何在 SELECT 子句中使用子查询来生成列表达式。我们讨论了一个简单的示例，并简要地介绍了

两个聚合函数，它们在你需要从另一个表中获取相关的汇总信息时很有用（下一章将详细介绍所有的聚合函数）。

然后讨论了如何在 WHERE 子句中使用子查询来创建复杂的筛选器。我们首先介绍了简单的比较，再简要地介绍了特殊的比较关键字 IN、SOME、ANY、ALL 和 EXISTS，在需要使用子查询来创建筛选器时，这些关键字很有用。

随后总结了子查询很有用的原因，并列举了一系列可使用子查询来解决的问题。最后，列举了一些子查询使用示例，这些示例被分成两组：在列表达式中使用子查询以及在筛选器中使用子查询。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

11.7　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**
  (1) 显示所有的商品及其最后一次被订购的日期。
  提示：使用聚合函数 MAX。
  解决方案见 CH11_Products_Last_Date（40 行）。你在结果集中看到日期为空的行了吗？你能解释出现这种情况的原因吗？
  (2) 列出订购了自行车的顾客。
  提示：使用 IN 创建一个筛选器。
  解决方案见 CH11_Customers_Ordered_Bikes_IN（23 行）。
  (3) 哪些商品从未被订购过？
  提示：使用 NOT IN 创建一个筛选器。
  解决方案见 CH11_Products_Not_Ordered（2 行）。
- **Entertainment Agency 数据库**
  (1) 列出所有的演唱组合及其签订的演出合约数量。
  提示：使用聚合函数 COUNT。
  解决方案见 CH11_Entertainer_Engagement_Count（13 行）。
  (2) 列出与演奏乡村音乐或乡村摇滚的演唱组合签约过的顾客。
  提示：使用 IN 创建一个筛选器。
  解决方案见 CH11_Customers_Who_Like_Country（13 行）。
  (3) 找出给顾客 Berg 或 Hallmark 演出过的演唱组合。
  提示：使用= SOME 创建一个筛选器。
  解决方案见 CH11_Entertainers_Berg_OR_Hallmark_SOME（8 行）。
  (4) 显示没有签订任何演出合约的经纪人。
  提示：使用 NOT IN 创建一个筛选器。
  解决方案见 CH11_Bad_Agents（1 行）。
- **School Scheduling 数据库**
  (1) 列出所有的教员及其讲授的课程数量。
  提示：使用聚合函数 COUNT。
  解决方案见 CH11_Staff_Class_Count（27 行）。
  (2) 显示注册了周二上课的课程的学生。
  提示：使用 IN 创建一个筛选器。
  解决方案见 CH11_Students_In_Class_Tuesdays（18 行）。

(3) 列出有课程在周三上课的科目。

提示：使用 IN 创建一个筛选器。

解决方案见 CH11_Subjects_On_Wednesday（34 行）。

- **Bowling League 数据库**

(1) 显示所有的投球手及其参加的比赛局数。

提示：使用聚合函数 COUNT。

解决方案见 CH11_Bowlers_And_Count_Games（32 行）。

(2) 列出原始得分比其球队中其他所有队员都低的投球手。

提示：使用< ALL 创建一个筛选器；另外，使用 DISTINCT，以防有投球手在多次比赛中的得分都一样低。

解决方案见 CH11_Bowlers_Low_Score（3 行）。

- **Recipes 数据库**

(1) 显示所有的菜品类型及其包含的菜品数量。

提示：使用聚合函数 COUNT。

解决方案见 CH11_Count_Of_Recipe_Types（7 行）。

(2) 列出在某个菜品中使用了且在该菜品中的度量单位不是默认度量单位的食材。

提示：使用<> SOME 创建一个筛选器。

解决方案见 CH11_Ingredients_Using_NonStandard_Measure（21 行）。

第四部分

数据汇总和分组

第 12 章 简单汇总

“有两种统计数据，一种是你查阅的，一种是你编造的。”

——摘自 Rex Stout 的悬疑小说 *Death of a Doxy*

本章涵盖如下主题：

- 聚合函数
- 在筛选器中使用聚合函数
- 语句举例
- 小结
- 练习

至此，你知道了如何根据需要选择列、定义表达式以添加更多细节、连接合适的表以提供所需的列，以及定义条件来筛选包含在结果集中的数据；换而言之，你具备了从一个或多个表中获取详细信息所需的全部技能。本章和接下来的两章将演示如何后退一步，从更广阔的角度审视数据，即眼观全局。

在本章中，你将学习如何使用聚合函数来生成基本的汇总信息；在第 13 章，你将学习如何在 SELECT 语句中使用 GROUP BY 子句将数据分组；在第 14 章，你将学习各种筛选分组数据的方法；而在第 21 章，你将学习如何成为“分组大师”。

12.1 聚合函数

本书前面的问题都要求从使用 FROM 和 WHERE 子句返回的行中获取各列的值，但你经常会遇到像下面这样的问题，而为了回答这些问题，只需根据多行中的值执行计算。

“有多少顾客住在西雅图？”

“在所有在售商品中，最高定价和最低定价分别是多少？”

“Mike Hernandez 负责讲授多少门课程？”

“最早的课程什么时间开始？”

“某门课程的平均时长是多少？”

“12 号订单的总金额是多少？”

SQL 标准提供了一组**聚合函数**，让你能够根据结果集中的行或值表达式返回的值计算出单个值。你可将函数应用于所有的行或值，也可通过使用 WHERE 子句将函数应用于一组特定的行或值。例如，你可以使用聚合函数来找出值表达式返回的一组值中的最小值或最大值，计算结果集包含的行数，或者计算值表达式返回的不同值的总和。图 12-1 说明了所有数据库系统都支持的基本聚合函数的语法。

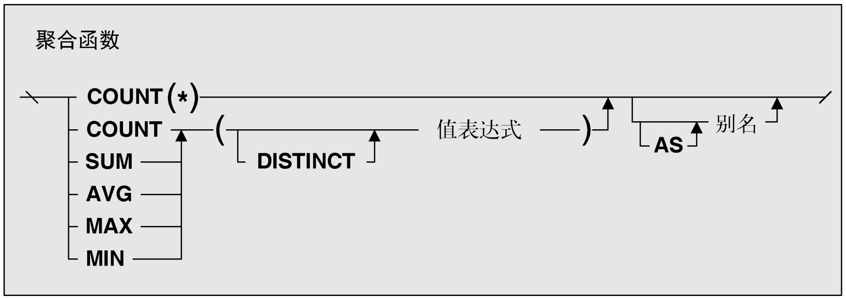


图 12-1　聚合函数的语法图

从中可知，聚合函数的语法简单而直观。前一章讨论了如何在子查询中使用两个聚合函数，在 SELECT 子句中返回单个计算得到的值，或将取回的计算得到的值用于 WHERE 子句的谓词中。本章将再列举几个这种用法的示例。

❖ **说明**　SQL: 2016 标准定义了十多个其他的聚合操作，但其中很多都没有在任何主要商业数据库系统中得到实现。本章只介绍所有主要数据库系统都支持的聚合函数，学习如何使用这些函数后，你可参阅你使用的数据库系统的文档，了解是否有其他可在 SQL 语句中使用的聚合函数。

每个聚合函数都返回单个值，不管它处理的是结果集中的行还是值表达式返回的值。除 COUNT(*)外，所有聚合函数都自动忽略 Null 值。你可在 SELECT 关键字后面的值表达式列表中同时使用多个聚合函数，甚至可混合使用包含聚合函数的值表达式和包含字面量值的值表达式，但一旦开始使用聚合表达式，就必须万分小心。

使用聚合表达式意味着让数据库系统根据一系列行计算出单个值。下一章你将学到，可使用 GROUP BY 子句来定义分组，但本章只介绍不显式指定分组的简单查询。在你没有指定分组方式时，数据库系统将使用 FROM 和 WHERE 子句返回的所有行来计算聚合表达式。

只要想一想就知道，在 SELECT 子句中包含聚合函数时，就不能再包含不在聚合函数内且引用了表列的值表达式。第 11 章介绍了聚合函数 COUNT 和 MAX。请看下面的 SQL 语句：

```
SQL          SELECT LastName, COUNT(*) As CountOfStudents
             FROM Students
```

上述语句包含 COUNT 函数，但没有指定分组方式，因此数据库系统将计算 FROM 子句返回的结果集中的行数。COUNT(*)返回单个值——Students 表的总行数，因此这个查询应返回一行。数据库系统该显示哪个 LastName 值呢？答案是它不知道该选择哪个，因此上述语句是非法的。

然而，可使用字面量表达式来进一步改善输出，这是因为字面量表达式是个常量，对所有行来说其值都相同。因此，下面的 SQL 完全合法：

```
SQL          SELECT 'The number of students is: ', COUNT(*)
                As CountOfStudents
             FROM Students
```

这条语句返回一行：

The number of students is:	18

指出一个小小的注意事项后，下面来介绍每个聚合函数以及如何使用它来解决问题。

12.1.1　使用 COUNT 计算行或值的个数

SQL 标准定义了两个版本的 COUNT 函数：COUNT(*)和 COUN(值表达式)，其中前者用于处理结果集中的行，而后者用于处理值表达式返回的值。

1. 计算总行数

要确定结果集包含多少行，可使用 COUNT(*)，它计算结果集中的总行数，包括重复的行以及包含 Null 值的行。下面的简单示例说明了可使用这个函数来解决的问题类型。

> ❖ **说明**　贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”流程。在所有的示例中，都假定你深入研究并掌握了本书前面介绍的概念，尤其是连接和子查询。

“告诉我我们公司总共有多少名员工。”

转换	Select the count of employees from the employees table（选择员工表中的所有行，并使用 COUNT 计算返回的总行数）
整理	Select ~~the~~ count ~~of employees~~ (*) from ~~the~~ employees ~~table~~
SQL	`SELECT COUNT(*)` `FROM Employees`

请注意，在整理结果中，我使用了(*)来指出我要计算 Employees 表中的总行数。解决这类问题时，你也应在整理结果中包含星号，这有助于确保你使用 COUNT 函数的正确版本。这条 SELECT 语句生成的结果集包含一个单列行，其中的数据值表示 Employees 表包含的总行数。

COUNT(*)函数能够处理的行数几乎不受限制。这里说“几乎”是因为大多数数据库系统使用一个整数来指出行数，这将行数限制设置成了 2 147 483 647。要告诉 COUNT(*)只考虑哪些行，可使用 WHERE 子句。例如，下面的 SELECT 演示了如何计算 Employees 表包含多少名居住在华盛顿州的员工：

SQL	`SELECT COUNT(*)`
	`FROM Employees`
	`WHERE EmpState = 'WA'`

在本章后面你将看到，可使用 WHERE 子句筛选出供聚合函数处理的行或值。

在 SELECT 语句中使用聚合函数时，在结果集中可能显示该函数返回的值的列名，也可能不显示。有些数据库系统提供默认列名，有些不提供，但你可使用 AS 选项在结果集中提供有意义的列名。下面演示了如何在前面的示例中使用这个选项：

SQL	`SELECT COUNT(*) AS TotalWashingtonEmployees`
	`FROM Employees`
	`WHERE EmpState = 'WA'`

现在结果集包含一个 TotalWashingtonEmployees 列，其中是函数 COUNT(*)返回的值。如图 12-1 的语法图所示，这种做法适用于所有聚合函数。

2. 计算列中值的个数或表达式返回的值的个数

可使用函数 COUNT（值表达式）来计算值表达式返回的非 Null 值的个数（这个函数版本通常简写为 COUNT，在本书余下的篇幅中，都将使用这种简写），它不忽略值表达式返回的任何值，而不管它们是不同的还是重复的，但自动排除所有 Null 值。可使用 COUNT 来回答类似于下面这样的问题：

“有多少顾客能够指出自己居住在哪个县？”

在这里，你需要确定 CustCounty 列有多少个实际值。前面说过，COUNT(*)将 Null 值也包含在内，无法提供正确答案，因此需要使用 COUNT，进而这样转换这个请求：

转换	Select the count of non-Null county values as NumberOfKnownCounties from the customers table（计算顾客表中 CustCounty 列不为 Null 的行数，并将结果作为 NumberOfKnownCounties 列返回）
整理	Select ~~the~~ count ~~of non-Null~~ (county) ~~values~~ as NumberOfKnownCounties from ~~the~~ customers ~~table~~

SQL	`SELECT COUNT(CustCounty)`
	`   AS NumberOfKnownCounties`
	`FROM Customers`

请注意，在转换和整理结果中，都明确地要求查找非 Null 值。虽然你知道 COUNT 只考虑非 Null 值，但最好明确指出这一点，确保最终使用正确的函数 COUNT。这条 SELECT 语句生成一行，其中包含一个数字值，指出在 CustCounty 列中找到了多少个非 Null 县名。大致而言，如果在这条 SELECT 语句中包含 `WHERE CustCounty IS NOT NULL`，并使用 `COUNT(*)`，将得到同样的结果。

前面说过，COUNT 函数将重复的值视为独特的，不将它们排除在最终的计数之外，但你可使用选项 DISTINCT 来将它们排除在外。下面的示例演示了如何使用这个选项。

“顾客表中有多少个不同的县名？”

转换	Select the count of unique non-Null county names as NumberOfUniqueCounties from the customers table（从顾客表中选择不为 Null 且独特的县名，计算选出的县名数量，并将其作为 NumberOfUniqueCounties）
整理	Select ~~the~~ count ~~of unique non-Null~~ (distinct county) ~~names~~ as NumberOfUniqueCounties from ~~the~~ customers ~~table~~

SQL	`SELECT COUNT(DISTINCT CustCounty)`
	`   AS NumberOfUniqueCounties`
	`FROM Customers`

当你使用了选项 DISTINCT 时，数据库系统将从 CustCounty 列检索所有非 Null 值，消除重复的值，再计算余下多少个值。当你在函数 SUM、AVG、MIN 或 MAX 中使用 DISTINCT 时，数据库系统的处理流程也几乎是这样的。

在下面的示例中，使用的请求与前一个有点不同，旨在说明可结合使用 COUNT 函数与筛选器。

“顾客表中有多少个属于俄勒冈州的不同县名？”

转换	Select the count of unique non-Null county names as NumberOfUniqueOregonCounties from the customers table where the state is ‘OR’（在顾客表中，从州为 OR 的行中选择不为 Null 的不同县名，再计算个数并将结果作为 NumberOfUniqueOregonCounties）
整理	Select ~~the~~ count ~~of unique non-Null~~ (distinct county) ~~names~~ as NumberOfUniqueOregonCounties from ~~the~~ customers ~~table~~ where ~~the~~ state ~~is~~ = ‘OR’

SQL	`SELECT COUNT(DISTINCT CustCounty)`
	`   AS NumberOfUniqueOregonCounties`
	`FROM Customers`
	`WHERE CustState = 'OR'`

必须指出的是，不能将 DISTINCT 用于 COUNT(*)，这种限制是有道理的，因为 COUNT(*)计算表中的总行数，而不管它们是否是重复的，也不管它们是否包含 Null 值。

12.1.2　使用 SUM 计算总计

可使用函数 SUM 来计算数字值表达式的总计，它处理值表达式返回的所有非 Null 值，并将累计值加入到结果集中。请注意，如果对于所有的行，值表达式的结果都为 Null，或者 FROM 和 WHERE 子句返回的结果集为空，SUM 将返回 Null。下面是一个可使用 SUM 来回答的问题：

“我给华盛顿州的员工支付的薪水总额是多少？”

转换	Select the sum of salary as TotalSalaryAmount from the employees table where the state is ‘WA’（从员工表中选择州为 WA 的员工，计算这些员工的薪水总额，并将结果作为 TotalSalaryAmount）
整理	Select ~~the~~ sum ~~of~~ (salary) as TotalSalaryAmount from ~~the~~ employees ~~table~~ where ~~the~~ state ~~is~~ = ‘WA’

SQL	`SELECT SUM(Salary) AS TotalSalaryAmount`
	`FROM Employees`
	`WHERE EmpState = 'WA'`

这里使用的值表达式是一个简单的列引用，但在 SUM 函数中，也可使用数字值表达式，如下面的示例所示。

“我们现有的存货价值多少？”

转换	Select the sum of wholesale price times quantity on hand as TotalInventoryValue from the products table（从商品表中选择批发价和库存量，用批发价乘以库存量计算每种商品的价值，再将各种商品的价值相加，并将结果作为 TotalInventoryValue）
整理	Select ~~the~~ sum ~~of~~ (wholesale price ~~times~~ * quantity on hand) as TotalInventoryValue from ~~the~~ products ~~table~~
SQL	SELECT SUM(WholesalePrice * QuantityOnHand) AS TotalInventoryValue FROM Products

你知道，仅当一行的 WholesalePrice 和 QuantityOnHand 列都包含实际值时，函数 SUM 才会处理它。在这里，数据库系统将处理 Products 表中所有符合条件的行，使用函数 SUM 累计结果，再将累计值加入到结果集中。

下面的示例演示了如何使用 SUM 来计算不同值的总和。

"计算我们销售的商品的不同批发价的总和。"

转换	Select the sum of unique wholesale costs as SumOfUniqueWholesaleCosts from the products table（从商品表中选择不同的批发价，计算它们的总和，并将结果作为 SumOfUniqueWholesaleCosts）
整理	Select ~~the~~ sum ~~of unique~~ (distinct wholesale costs) as SumOfUniqueWholesaleCosts from ~~the~~ products ~~table~~
SQL	SELECT SUM(DISTINCT WholesaleCost) AS SumOfUniqueWholesaleCosts FROM Products

12.1.3　使用 AVG 计算平均值

另一个可用于对数字值执行计算的函数是 AVG，它计算值表达式返回的所有非空值的算术平均值。你可使用 AVG 来回答类似于下面这样的问题：

"10014 号供应商的平均合约价格是多少？"

转换	Select the average of contract price as AverageContractPrice from the vendor contracts table where the vendor ID is 10014（在供应商表中，从供应商 ID 为 10014 的行中选择合约价格，计算它们的平均值，并将结果作为 AverageContractPrice）
整理	Select ~~the average of~~ avg (contract price) as AverageContractPrice from ~~the~~ vendor contracts ~~table~~ where the vendor ID ~~is~~ = 10014
SQL	SELECT AVG(ContractPrice) AS AverageContractPrice FROM Vendor_Contracts WHERE VendorID = 10014

在整理结果中，务必划掉单词 average，并用 avg 取而代之，以免你不小心在 SELECT 子句中使用 Average。Average 不是合法的 SQL 关键字，因此如果你使用它，SELECT 语句将无法执行。

像 SUM 函数一样，AVG 也可用来处理数值表达式。别忘了，在 AVG 中，不能指定结果不是数字的值表达式；如果将函数 SUM 或 AVG 用于处理字符串或日期时间数据，大多数数据库系统会报错。

"在 64 号订单中，各种商品的平均总价是多少？"

转换	Select the average of price times quantity ordered as AverageItemTotal from the order details table where order ID is 64（在订单详情表中，从订单 ID 为 64 的行中选择价格和订购数量，计算对应的价格和订购数量的乘积，再计算这些乘积的平均值，将结果作为 AverageItemTotal）
整理	Select ~~the average of~~ avg (price ~~times~~ * quantity ordered) as AverageItemTotal from ~~the~~ order details ~~table~~ where order ID ~~is~~ = 64
SQL	SELECT AVG(Price * QuantityOrdered) AS AverageItemTotal FROM Order_Details WHERE OrderID = 64

别忘了，仅当一行的 Price 和 QuantityOrdered 列都包含实际值时，AVG 函数才会处理它；如果不满足这个条件，数值表达式的结果将为 Null，而 AVG 函数将忽略该行。与 SUM 一样，如果对于所有的行，指定值表达式的结果都为

Null，或者 FROM 和 WHERE 返回的结果集为空，AVG 将返回一个 Null 值。

在下面的示例中，我使用了选项 DISTINCT 来计算不同数值的平均值：

“计算各种不同商品价格的平均值。”

转换	Select the average of unique prices as UniqueProductPrices from the products table（从商品表中选择不同的价格，计算它们的平均值，并将结果作为 UniqueProductPrices）
整理	Select ~~the average of unique~~ avg (distinct prices) as UniqueProductPrices from ~~the~~ products ~~table~~

SQL	`SELECT AVG(DISTINCT Price)`
	`   AS UniqueProductPrices`
	`FROM Products`

12.1.4　使用 MAX 找出最大值

可使用函数 MAX 来找出值表达式返回的最大值，这个函数能够处理任何类型的数据，它返回的值取决于被处理的数据。

- ❑ 字符串：MAX 返回的值取决于数据库系统或计算机使用的排序序列。例如，数据库系统使用的是 ASCII 字符集且不区分大小写，它将按如下顺序排列公司名：. . . 4th Dimension Productions . . . Al’s Auto Shop . . . allegheny & associates . . . Zercon Productions . . . zorn credit services。在这个示例中，MAX 将把 zorn credit services 作为最大值返回。
- ❑ 数字：MAX 返回最大的数字。
- ❑ 日期时间：MAX 按时间顺序排列日期和时间，并返回最近（最后）的日期或时间。

下面列举两个可使用 MAX 来回答的问题。

“最高的演出合约价格是多少？”

转换	Select the maximum contract price as LargestContractPrice from the engagements table（从演出合约表中选择合约价格，找出其中最大的值，并将其作为 LargestContractPrice）
整理	Select ~~the~~ max~~imum~~ (contract price) as LargestContractPrice from ~~the~~ engagements ~~table~~

SQL	`SELECT MAX(ContractPrice)`
	`   AS LargestContractPrice`
	`FROM Engagements`

“在 3314 号订单中，最高的商品总价是多少？”

转换	Select the maximum price times quantity ordered as LargestItemTotal from the order details table where the order ID is 3314（在订单详情表中，选择订单 ID 为 3314 的行，计算各行中价格与订购数量的乘积，在这些乘积中找出最大的值，并将其作为 LargestItemTotal）
整理	Select ~~the~~ max~~imum~~ (price ~~times~~ * quantity ordered) as LargestItemTotal from ~~the~~ order details ~~table~~ where ~~the~~ order ID ~~is~~ = 3314

SQL	`SELECT MAX(Price * QuantityOrdered)`
	`   AS LargestItemTotal`
	`FROM Order_Details`
	`WHERE OrderID = 3314`

你可能想使用 DISTINCT 来返回单个最高或最近的值。虽然 SQL 标准将 DISTINCT 作为函数 MAX 的一个选项，但 DISTINCT 对 MAX 函数没有任何影响。最大值只有一个，不管它是否是独一无二的。例如，如果要在 Agents 表中查找最近的聘用日期，下面两个表达式返回的值相同：

```
SELECT MAX(DateHired) FROM Agents
SELECT MAX(DISTINCT DateHired) FROM Agents
```

这里之所以列出这两个版本，是因为它们都符合最新的 SQL 标准，但我推荐使用不包含 DISTINCT 选项的版本。

包含 DISTINCT 选项意味着让数据库系统先做额外而不必要的工作——找出不同的值，再确定哪个是最大或最近的。

12.1.5　使用 MIN 找出最小值

利用函数 MIN 能够确定值表达式返回的最小值，其工作原理与 MAX 相同，但返回相反的值：第一个字符串（基于排序序列）、最小的数字和最早的日期或时间。

使用函数 MIN 可回答类似于下面这样的问题：

"最低的商品价格是多少？"

转换	Select the minimum price as LowestProductPrice from the products table（从商品表中选择价格，并找出其中最小的值，将其作为 LowestProductPrice）
整理	Select ~~the~~ min~~imum~~ (price) as LowestProductPrice from ~~the~~ products ~~table~~
SQL	`SELECT MIN(Price) AS LowestProductPrice` `FROM Products`

"在 3314 号订单中，最低的商品总价是多少？"

转换	Select the minimum price times quantity ordered as LowestItemTotal from the order details table where the order ID is 3314（在订单详情表中，选择订单 ID 为 3314 的行，计算各行中价格与订购数量的乘积，然后在这些乘积中找出最小的值，并将其作为 LowestItemTotal）
整理	Select ~~the~~ min~~imum~~ (price ~~times~~ * quantity ordered) as LowestItemTotal from ~~the~~ order details ~~table~~ where ~~the~~ order ID ~~is~~ = 3314

SQL	`SELECT MIN(Price * QuantityOrdered)`
	`   AS LowestItemTotal`
	`FROM Order_Details`
	`WHERE OrderID = 3314`

必须指出的是，DISTINCT 选项对函数 MIN 没有任何影响（你知道，函数 MAX 的情况也如此），因为只能有一个最小值，不管它是否是独一无二的。例如，下面两个表达式返回的值相同：

```
SELECT MIN(DateHired) FROM Agents
SELECT MIN(DISTINCT DateHired) FROM Agents
```

这里之所以列出这两个版本，是因为它们都符合最新的 SQL 标准，但在前面介绍 MAX 是说过，推荐使用不包含 DISTINCT 选项的版本。包含 DISTINCT 选项意味着让数据库系统先做额外而不必要的工作——找出不同的值，再确定哪个是最小或最早的。

12.1.6　使用多个函数

本节开头说过，可同时使用多个聚合函数，这让你只需使用一条 SELECT 语句就可显示形成鲜明对比的信息。例如，可使用函数 MIN 和 MAX 来显示特定顾客第一次和最后一次下单的日期，还可使用函数 MAX、MIN 和 AVG 显示特定学生的最好成绩、最差成绩和平均成绩。下面再举几个同时使用多个聚合函数的示例。

"告诉我广告部员工的最早和最晚的述职日期。"

转换	Select the minimum review date as EarliestReviewDate and the maximum review date as RecentReviewDate from the employees table where the department is 'Advertising'（在员工表中，从部门为 Advertising 的行中选择述职日期，再在这些日期中找出最早和最晚的，将它们分别作为 EarliestReviewDate 和 RecentReviewDate）
整理	Select ~~the~~ min~~imum~~ review date as EarliestReviewDate, ~~and the~~ max~~imum~~ review date as RecentReviewDate from ~~the~~ employees ~~table~~ where ~~the~~ department ~~is~~ = 'Advertising'

SQL	`SELECT MIN(ReviewDate) AS EarliestReviewDate,`
	`   MAX(ReviewDate) AS RecentReviewDate`
	`FROM Employees`
	`WHERE Department = 'Advertising'`

“553 号订单包含多少种不同的商品？该订单的总价是多少？”

转换	Select the unique count of product ID as TotalProductsPurchased and the sum of price times quantity ordered as OrderAmount from the order details table where the order number is 553（在订单详情表中，从订单号为 553 的行中选择商品 ID、价格和订购数量；计算有多少个不同的商品 ID，并将其作为 TotalProductsPurchased；计算各行中价格与订购数量的乘积，并将这些乘积累积，将结果作为 OrderAmount）
整理	Select ~~the unique~~ count ~~of~~ (DISINCT product ID) as TotalProductsPurchased, ~~and the~~ sum ~~of~~ (price ~~times~~ * quantity ordered) as OrderAmount from ~~the~~ order details ~~table~~ where ~~the~~ order number ~~is~~ = 553
SQL	SELECT COUNT(DISTINCT ProductID) AS TotalProductsPurchased, SUM(Price * QuantityOrdered) AS OrderAmount FROM Order_Details WHERE OrderNumber = 553

千万别忘了，使用多个聚合函数时，有两个限制。一是不能在一个聚合函数中嵌入另一个聚合函数（唯一的例外是，你是在窗口函数中使用它们，详情请参阅第 22 章）。由于这种限制，下面的表达式非法：

```
SUM(AVG(LineItemTotal))
```

二是不能将子查询用作聚合函数的值表达式。例如，由于这种限制，下面的表达式非法：

```
AVG((SELECT Price FROM Products WHERE Category = 'Bikes'))
```

虽然存在这些限制，但你可在 SELECT 子句中使用聚合函数轻松地获取相对复杂的统计信息。下面来看看如何使用聚合函数来筛选结果集中的信息。

12.2 在筛选器中使用聚合函数

由于聚合函数返回单个值，因此可在查找条件中的比较谓词中使用它。然而，你必须将聚合函数放在子查询中，再在比较谓词中使用该子查询。如果你觉得这听起来有点耳熟，这种感觉完全正确。在第 11 章，你学习了如何在 WHERE 子句的查找条件中使用子查询，以及如何在子查询中使用聚合函数，因此你已大致知道如何使用聚合函数来筛选出要加入到结果集中的数据。下面来详细介绍这一点。

通过在比较谓词中使用聚合函数，可将值表达式的值与单个统计值进行比较。虽然使用字面量值也可完成这种任务，但子查询提供了更大的灵活性，让比较条件的动态性更强。例如，假设你要向数据库发出如下请求：

“列出零售价不高于平均零售价的商品。”

为解决这个问题，一种办法是先手动计算平均零售价，再在比较谓词中使用这个值。

转换	Select the product name from the products table where the retail price is less than or equal to $196.03（在商品表中，从零售价不高于 196.03 美元的行中选择商品名）
整理	Select ~~the~~ product name from ~~the~~ products ~~table~~ where ~~the~~ retail price ~~is less than or equal to~~ <= $196.03
SQL	SELECT ProductName FROM Products WHERE RetailPrice <= 196.03

为何要去做不必要的工作呢？你可在一个子查询中使用聚合函数，让数据库系统替你去完成工作呀！

转换	Select the product name from the products table where the retail price is less than or equal to the overall average retail price in the products table（在商品表中，从零售价不高于平均零售价的行中选择商品名）
整理	Select ~~the~~ product name from ~~the~~ products ~~table~~ where ~~the~~ retail price ~~is less than or equal to the~~ <= ~~overall~~ (select ~~average~~ avg retail price ~~in the~~ from products ~~table~~)
SQL	SELECT ProductName FROM Products WHERE RetailPrice <= (SELECT AVG(RetailPrice) FROM Products)

我将这个查询保存到了示例数据库 Sales Orders 中，名为 CH12_Products_LTE_Avg_Price。

显然，使用子查询和聚合函数是最佳的选择。如果你使用字面量值，就必须在每次执行 SELECT 语句前重新计算平均价，以防有商品的零售价发生了变化；然后，你必须确保在比较谓词中输入正确的值。但如果你使用子查询，就根本不用担心这些问题。每当你执行 SELECT 语句时，都将执行其中的 AVG 函数，而它总是返回正确的值，不管你是否修改了商品的零售价（对于你在子查询中使用的其他任何聚合函数，情况也如此）。

你可在子查询中使用 WHERE 子句限制将被聚合函数处理的行，这样能够缩小聚合函数返回的统计值覆盖的范围。你在第 11 章已经学过如何在子查询中使用 WHERE 子句，因此下面来看一个如何使用这种技巧的示例。

“列出这样的演出合约的编号和合约价格，即其合约价格高于 2017 年 9 月份所有演出合约的合约价格总和。”

转换	Select engagement number and contract price from the engagements table where the contract price is greater than the sum of all contract prices of engagements dated between September 1, 2017, and September 30, 2017（计算演出日期在 2017 年 9 月 1 日和 9 月 30 日之间的所有演出合约的合约价格总和；在演出合约表中选择合约价格高于前述价格总和的演出合约的编号和合约价格）
整理	Select engagement number, ~~and~~ contract price from ~~the~~ engagements ~~table~~ where ~~the~~ contract price ~~is greater than~~ > ~~the~~ (select sum ~~of all~~ (contract prices) from engagements where ~~dated~~ start date between ~~September 1, 2017,~~ ‘2017-09-01’ and ~~September 30, 2017~~ ‘2017-09-30’)

SQL	`SELECT EngagementNumber, ContractPrice`
	`FROM Engagements`
	`WHERE ContractPrice >`
	`   (SELECT SUM(ContractPrice) FROM Engagements`
	`     WHERE StartDate BETWEEN '2017-09-01'`
	`     AND '2017-09-30')`

我将这个查询保存在了示例数据库 Entertainment Agency 中，并将其命名为 CH12_Engagements_GT_SUM_September。

你可能发现，需要在筛选器中使用聚合函数的情况很少，但需要回答那些异乎寻常的问题时，它们确实能够派上用场。

12.3　语句举例

本章介绍了如何在 SELECT 子句中使用聚合函数，还有如何在用作比较谓词一部分的子查询中使用它们。下面来看一些聚合函数使用示例，它们使用了各个示例数据库中的表，演示了如何使用聚合函数来生成输出列以及如何在子查询中使用它们。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库，查询名以 CH12 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明**　别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录 B。这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。

- **Sales Orders 数据库**

“我们在加州有多少顾客？”

转换/整理	Select ~~the~~ count(*) as NumberOfCACustomers ~~of all customers~~ from ~~the~~ customers ~~table~~ where ~~the~~ state ~~is~~ = ‘CA’（从顾客表中选择州为 CA 的所有顾客，使用 count(*)计算顾客数量，并将其作为 NumberOfCACustomers）

SQL

```
SELECT COUNT(*) AS NumberOfCACustomers
FROM Customers
WHERE CustState = 'CA'
```

CH12_Number_Of_California_Customers（1 行）

NumberOfCACustomers
7

“列出商品表中报价不低于平均零售价的商品的名称和编号。”

转换/整理

Select ~~the~~ product name, ~~and the~~ product number from ~~the~~ products ~~table~~ inner join~~ed with the~~ order details ~~table~~ on products.product number ~~in the products table matches~~ = order_details.product number ~~in the order details table~~ where ~~the~~ quoted price ~~is greater than or equal to~~ >= (select ~~the average~~ avg(retail price) ~~in the~~ from products ~~table~~)（从商品表中选择零售价并使用 avg 计算它们的平均值；基于商品编号相同将商品表内连接到订单详情表，从报价比低于前述平均值的行中选择商品的名称和编号）

SQL

```
SELECT DISTINCT Products.ProductName,
   Products.ProductNumber
FROM Products
INNER JOIN Order_Details
ON Products.ProductNumber =
   Order_Details.ProductNumber
WHERE Order_Details.QuotedPrice >=
   (SELECT AVG(RetailPrice)
   FROM Products)
```

❖ **说明** 我使用了 DISTINCT 以便只返回不同的商品，因为有些商品可能被订购了多次，但对于每种商品的名称及其编号，我只想显示一次。

CH12_Quoted_Price_vs_Average_Retail_Price（4 行）

ProductName	ProductNumber
Eagle FS-3 Mountain Bike	2
GT RTS-2 Mountain Bike	11
Trek 9000 Mountain Bike	1
Viscount Mountain Bike	6

- **Entertainment Agency 数据库**

“列出最早的演出合约的编号和合约价格。”

转换/整理

Select engagement number, ~~and~~ contract price from ~~the~~ engagements ~~table~~ where ~~the~~ start date ~~is equal to the~~ = ~~earliest~~ (select min(start date) ~~in the~~ from engagements ~~table~~)（从演出合约表中选择开始日期，并使用 min 找出最早的那个；从演出合约表中选择开始日期与前面找出的日期相同的演出合约的编号和合约价格）

SQL

```
SELECT EngagementNumber, ContractPrice
FROM Engagements
WHERE StartDate =
   (SELECT MIN(StartDate) FROM Engagements)
```

CH12_Earliest_Contracts（1 行）

EngagementNumber	ContractPrice
2	$200.00

"2017 年 10 月所有演出合约的总价是多少？"

转换/整理　Select ~~the~~ sum ~~of~~ (contract price) as TotalBookedValue from ~~the~~ engagements ~~table~~ where ~~the~~ start date ~~is~~ between ~~October 1, 2017~~ '2017-10-01' and ~~October 31, 2017~~ '2017-10-31'（在演出合约表中，选择开始日期在 2017 年 10 月 1 日与 10 月 31 日之间的演出合约的合约价格，并使用 sum 计算这些价格的总和，将结果作为 TotalBookedValue）

SQL

```sql
SELECT SUM(ContractPrice) AS TotalBookedValue
FROM Engagements
WHERE StartDate
   BETWEEN '2017-10-01' AND '2017-10-31'
```

CH12_Total_Booked_Value_For_October_2017（1 行）

TotalBookedValue
$30,125.00

- **School Scheduling 数据库**

"我们给教员支付的最高薪水是多少？"

转换/整理　Select ~~the~~ max~~imum~~ (salary) as LargestStaffSalary from ~~the~~ staff ~~table~~（从教员表中选择薪水，使用 max 找出其中最大的值，将其作为 LargestStaffSalary）

SQL

```sql
SELECT Max(Salary) AS LargestStaffSalary
FROM Staff
```

CH12_Largest_Staff_Salary（1 行）

LargestStaffSalary
$60,000.00

"我们给加州的教员支付的薪水总额是多少？"

转换/整理　Select ~~the~~ sum ~~of~~ (salary) as TotalAmountPaid from ~~the~~ staff ~~table for all our California staff~~ where state = 'CA'（在教员表中，从州为 CA 的行中选择薪水，并使用 sum 计算这些薪水的总和，将结果作为 TotalAmountPaid）

SQL

```sql
SELECT SUM(Salary) AS TotalAmountPaid
FROM Staff
WHERE StfState = 'CA'
```

CH12_Total_Salary_Paid_To_California_Staff（1 行）

TotalAmountPaid
$209,000.00

- **Bowling League 数据库**

"有多少次联赛是在保龄球馆 Red Rooster Lanes 举行的？"

转换/整理　Select ~~the~~ count ~~of~~ (tourney location)~~s~~ as NumberOfTournaments from ~~the~~ tournaments ~~table~~ where ~~the~~ tourney location ~~is~~ = 'Red Rooster Lanes'（在联赛表中，选择联赛地点为 Red Rooster Lanes 的联赛地点，然后使用 count 计算联赛地点数，并将结果作为 NumberOfTournaments）

SQL

```sql
SELECT COUNT(TourneyLocation)
   AS NumberOfTournaments
FROM Tournaments
WHERE TourneyLocation = 'Red Rooster Lanes'
```

❖ **说明**　由于这个查询是根据 TourneyLocation 进行筛选的，因此也可使用 COUNT(*)。

CH12_Number_Of_Tournaments_At_Red_Rooster_Lanes（1 行）

NumberOfTournaments
3

“列出个人平均得分不低于总体平均得分的投球手的姓和名，并按字母顺序排列。”

转换/整理

Select ~~the~~ last name, ~~and~~ first name from ~~the~~ bowlers ~~table~~ where ~~the~~ (select ~~average~~ avg(raw score) from ~~the~~ bowlers scores ~~table~~ as BS ~~for the current bowler~~ where BS.bowler ID = bowlers.bowler ID) ~~is greater than or equal to the~~ >= ~~overall~~ (select avg(raw score) ~~score in the~~ from bowler scores ~~table~~) ~~sorted~~ order by last name, ~~and~~ first name［从投球手得分表中选择原始得分，使用 avg 计算这些得分的平均值；从投球手表中选择姓和名，生成一个结果集；对于其中的每一行做这样的处理，即基于保龄球 ID 相同从投球手表（别名为 BS）中选择该行表示的投球手的原始得分，并使用 avg 计算这些得分的平均值，如果结果不小于前面计算得到的平均值，就返回这一行；依次根据姓和名对返回的行进行排序］

SQL

```
SELECT Bowlers.BowlerLastName,
   Bowlers.BowlerFirstName
FROM Bowlers
WHERE (SELECT AVG(RawScore)
FROM Bowler_Scores AS BS
WHERE BS.BowlerID = Bowlers.BowlerID)
>=(SELECT AVG(RawScore) FROM Bowler_Scores)
ORDER BY Bowlers.BowlerLastName,
         Bowlers.BowlerFirstName
```

CH12_Better_Than_Overall_Average（17 行）

BowlerLastName	BowlerFirstName
Cunningham	David
Fournier	David
Hallmark	Alaina
Hallmark	Gary
Hernandez	Michael
Kennedy	Angel
Kennedy	John
Patterson	Kathryn
Patterson	Neil
Patterson	Rachel
Clothier	Ben
Thompson	Mary
Thompson	Sarah
Thompson	William
Viescas	Caleb
Viescas	David
Viescas	John

❖ **说明**　如你所见，这里在 WHERE 子句创造性地使用了两个子查询来解决这个问题。

- **Recipes 数据库**

“有多少菜品使用了牛肉这种食材？”

转换/整理

Select ~~the~~ count(*) ~~of recipes~~ as NumberOfRecipes from ~~the~~ recipes ~~table~~ where ~~the~~ recipe ID ~~is~~ in ~~the~~ (select~~ion of~~ recipe ID~~s in the~~ from recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on recipe_ingredients.ingredient ID ~~in the recipe ingredients table matches~~ = ingredients. ingredient ID ~~in the ingredients table~~ where ~~the~~ ingredient name ~~is~~ like ‘%Beef%’)（基于食材 ID 相同将菜品食材表内连接到食材表，从食材名包含 Beef 的行中选择菜品 ID，生成一个菜品 ID 列表；在菜品表中，选择菜品 ID 位于前述列表中的行，使用 count(*)计算得到的行数，并将结果作为 NumberOfRecipes）

SQL

```
SELECT COUNT(*) AS NumberOfRecipes
FROM Recipes
WHERE Recipes.RecipeID IN
   (SELECT RecipeID
   FROM Recipe_Ingredients
   INNER JOIN Ingredients ON
      Recipe_Ingredients.IngredientID =
      Ingredients.IngredientID
   WHERE Ingredients.IngredientName
     LIKE '%Beef%')
```

CH12_Recipes_With_Beef_Ingredient（1 行）

NumberOfRecipes
3

“有多少种食材的度量单位为杯？”

转换/整理　Select ~~the~~ count (*) ~~of ingredients~~ as NumberOfIngredients from ~~the~~ ingredients ~~table~~ inner join~~ed with the~~ measurements ~~table~~ on ingredients.measure amount ID ~~in the ingredients table matches~~ = measurements.measure amount ID ~~in the measurements table~~ where ~~the~~ measurement description ~~is~~ = ‘Cup’（基于度量单位 ID 相同将食材表内连接到度量单位表，选择度量单位描述为 Cup 的行，使用 count(*)计算得到的行数，将结果作为 NumberOfIngredients）

SQL

```
SELECT COUNT(*) AS NumberOfIngredients
FROM Ingredients
INNER JOIN Measurements
ON Ingredients.MeasureAmountID =
   Measurements.MeasureAmountID
WHERE MeasurementDescription = 'Cup'
```

CH12_Number_of_Ingredients_Measured_by_the_Cup（1 行）

NumberOfIngredients
12

12.4　小结

本章首先简要地介绍了聚合函数。你学习了 5 个不同的函数，并了解到可在 SELECT 语句的 SELECT 和 WHERE 子句中使用它们。你还了解到，除 COUNT(*)外的所有聚合函数都不考虑 Null 值。

接下来介绍了如何使用各个聚合函数。你学习了如何使用 COUNT 函数计算行或值的个数，如何使用函数 MAX 和 MIN 找出最大值和最小值，如何使用函数 AVG 计算平均值，以及如何使用函数 SUM 计算一组值的总和。此外还介绍了如何在每个函数中使用 DISTINCT 选项，并指出这个选项对函数 MAX 和 MIN 没有影响。

最后介绍了如何在筛选器中使用聚合函数。你了解到，可在子查询中使用聚合函数，再将该子查询作为筛选器的一部分。你还了解到，还可将筛选器应用于子查询，让聚合函数根据一组特定的数据来计算结果。

使用聚合函数可做的事情很多，本章介绍的只是冰山一角。下一章将介绍如何将聚合函数应用于分组数据，以提供更精密的统计信息，以及如何对聚合计算应用筛选器。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

12.5　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 山地自行车的平均零售价是多少？

  解决方案见 CH12_Average_Price_Of_A_Mountain_Bike（1 行值：$1321.25）。

(2) 最后一个订单是在哪天下的？

解决方案见 CH12_Most_Recent_Order_Date（1 行值：2018-03-01）。

(3) 8 号订单的总价是多少？

解决方案见 CH12_Total_Amount_For_Order_Number_8（1 行值：$1492.60）。

- **Entertainment Agency 数据库**

(1) 经纪人的平均薪水是多少？

解决方案见 CH12_Average_Agent_Salary（1 行值：$24850.00）。

(2) 列出合约价格低于平均合约价格的所有演出合约的编号。

提示：要解决这个问题，需要使用子查询。

解决方案见 CH12_Contract_Price_GE_Average_Contract_Price（43 行）。

(3) 有多少演唱组合住在 Bellevue？

解决方案见 CH12_Number_Of_Bellevue_Entertainers（1 行值：3）。

(4) 在 2017 年 10 月份，最先履行的演出合约是哪些？

解决方案见 CH12_Earliest_October_Engagements（3 行）。

- **School Scheduling 数据库**

(1) 当前课程的平均时长是多少？

解决方案见 CH12_Average_Class_Duration（1 行值：78.939；但 SQL Server 将结果截断为 78）。

(2) 列出从最早的聘用日期起就一直在学校工作的教员的姓和名。

提示：需要使用一个子查询，并在其中使用一个聚合函数来评估 DateHired 列。

解决方案见 CH12_Most_Senior_Staff_Members（1 行值：Alborous, Sam）。

(3) 有多少课程是在 3346 教室上的？

解决方案见 CH12_Number_Of_Classes_Held_In_Room_3346（1 行值：10）。

- **Bowling League 数据库**

(1) 当前最高的让分是多少？

解决方案见 CH12_Current_Highest_Handicap（1 行值：52）。

(2) 最早的联赛是在什么地方举行的？

解决方案见 CH12_Tourney_Locations_For_Earliest_Date（1 行值：Red Rooster Lanes）。

(3) 已安排的最后一次联赛在哪天举行？

解决方案见 CH12_Last_Tourney_Date（1 行值：2018-08-16）。

- **Recipes 数据库**

(1) 哪个菜品需要的大蒜瓣（cloves）数最多？

提示：要解决这个问题，需要使用内连接和子查询。请注意，在所有菜品中，大蒜的度量单位都是瓣，即 cloves，因此不需要根据这个度量单位进行筛选。

解决方案见 CH12_Recipe_With_Most_Cloves_of_Garlic（1 行值：Roast Beef）。

(2) 计算属于主菜的菜品数量？

提示：要找出所有的主菜，需要将 Recipe_Classes 表连接到 Recipes 表，但你也可投机取巧，查找 RecipeClassID 为 1 的菜品。

解决方案见 CH12_Number_Of_Main_Course_ Recipes（1 行值：7）。

(3) 计算所有菜品总共需要多少茶匙盐。

提示：在所有菜品中，盐的度量单位都是茶匙，因此无须根据这个度量单位进行筛选。

解决方案见 CH12_Total_Salt_Used（1 行值：8.75）。

第 13 章

数据分组

13

“不要死抠细微末节，要放眼全局。”

——协约国军队总司令斐迪南·福煦元帅

本章涵盖如下主题：

- 为何要将数据分组
- GROUP BY 子句
- 一些限制
- GROUP BY 的用途
- 语句举例
- 小结
- 练习

第 12 章阐述了如何使用聚合函数（COUNT、MIN、MAX、AVG 和 SUM）让数据库系统根据 FROM 和 WHERE 子句指定的表中所有的行计算出单个值，同时指出：在 SELECT 子句中包含使用了聚合函数的值表达式后，SELECT 子句中的其他值表达式只能是字面量或包含聚合函数。在需要根据结果集计算出单个值时，聚合函数很有用，但如果要计算一系列小计，该怎么办呢？本章将介绍如何通过将数据分组来计算小计，而第 21 章将介绍如何计算更复杂的小计。

13.1　为何要将数据分组

使用数据库 Sales Orders 时，能够找出订单数（COUNT）、总销量（SUM）、平均销量（AVG）、最小订单（MIN）或最大订单（MAX）确实很有用。如果要按顾客、下单日期或商品计算这些值，可使用筛选器（WHERE）获取与特定顾客或商品相关的行，但如果要计算每位顾客的小计并与小计一起显示顾客姓名呢？为此可让数据库系统将行**分组**。

同理，在数据库 Entertainment Agency 中，找出合约数、合约价格总额、最低合约价格、最高合约价格很容易。还可通过使用筛选器来计算特定演唱组合、特定顾客或特定时段的上述值。同样，如果要计算每位顾客或演唱组合的小计，必须对行进行分组。

开始有点明白了吧？让数据库系统根据列值将行分组时，它将根据不同的列值建立多个子集，这样你就可计算每组的聚合值。来看一个基于数据库 Entertainment Agency 的简单示例。首先需要编写一个查询，取回所需的列——演唱组合和合约价格，相应的 SQL 如下：

```
SQL          SELECT Entertainers.EntStageName,
                Engagements.ContractPrice
             FROM Entertainers
             INNER JOIN Engagements
             ON Entertainers.EntertainerID =
                Engagements.EntertainerID
             ORDER BY EntStageName
```

结果类似于下面这样（在这个示例数据库中，我保存了这个查询并将其命名为 CH13_Entertainers_And_ContractPrices）：

EntStageName	ContractPrice
Carol Peacock Trio	$140.00
Carol Peacock Trio	$1,670.00
Carol Peacock Trio	$770.00
Carol Peacock Trio	$1,670.00
Carol Peacock Trio	$1,670.00
Carol Peacock Trio	$320.00
Carol Peacock Trio	$1,400.00
Carol Peacock Trio	$680.00
Carol Peacock Trio	$410.00
Carol Peacock Trio	$1,940.00
Carol Peacock Trio	$410.00
Caroline Coie Cuartet	$1,250.00
Caroline Coie Cuartet	$2,450.00
Caroline Coie Cuartet	$1,490.00
Caroline Coie Cuartet	$1,370.00
<< 其他行 >>	

你知道，可计算行数，还可找出最低 ContractPrice 列的最小值、最大值、平均值或总和——条件是将 EntStageName 列排除在外。但如果要让数据库系统根据这列分组，可将其保留。当你要求根据艺名分组时，数据库系统将把整个表分成多组，其中第一个分组包含开头 11 行（Carol Peacock Trio），第二个分组包含接下来的 11 行（Caroline Coie Cuartet），以此类推。这样就可计算各组的行数以及 ContractPrice 列的总和、最小值、最大值或平均值，结果是每组一个聚合行，类似于下面这样：

EntStageName	NumContracts	TotPrice	MinPrice	MaxPrice	AvgPrice
Carol Peacock Trio	11	$11,080.00	$140.00	$1,940.00	$1,007.27
Caroline Coie Cuartet	11	$15,070.00	$290.00	$2,450.00	$1,370.00
Coldwater Cattle Company	8	$14,875.00	$350.00	$3,800.00	$1,859.38
Country Feeling	15	$34,080.00	$275.00	$14,105.00	$2,272.00
Jazz Persuasion	7	$5,480.00	$500.00	$1,670.00	$782.86
Jim Glynn	9	$3,030.00	$110.00	$770.00	$336.67
Julia Schnebly	8	$4,345.00	$275.00	$875.00	$543.13
JV & the Deep Six	10	$17,150.00	$950.00	$3,650.00	$1,715.00
Modern Dance	10	$14,600.00	$650.00	$2,930.00	$1,460.00
Saturday Revue	9	$11,550.00	$290.00	$2,930.00	$1,283.33
Susan McLain	6	$2,670.00	$230.00	$800.00	$445.00
Topazz	7	$6,620.00	$590.00	$1,550.00	$945.71
<< 其他行 >>					

看起来很有趣，不是吗？我敢肯定，你现在很想知道如何做到这一点。接下来的几节将介绍其中的所有细节。

❖ **说明**　本书前言提醒过，在你使用的数据库系统中，行的排列顺序不一定与本书示例中显示的相同，除非使用 ORDER BY 子句。即便指定了排列方式，在数据库系统返回的结果中，对于 ORDER BY 子句中未包含的列，根据它们的排列顺序也可能不同，这是因为优化方法因数据库系统而异。

如果你在 Microsoft SQL Server 中运行示例，则从视图中选择行时，将不会遵循其中的 ORDER BY 子句指定的排列顺序。要遵循 ORDER BY 子句指定的排列顺序，必须打开视图的设计，并在显示界面中执行它。

13.2　GROUP BY 子句

第 12 章介绍过，使用聚合函数可计算出各种有趣的信息，但你可能注意到了，在该章列举的所有示例中，聚合函数都是根据 FROM 和 WHERE 子句返回的所有行进行计算的。你可使用 WHERE 子句来筛选结果集，得到特定的分组，但无法使用单个查询得到多个分组。要使用单个查询实现这种分组汇总，需要使用另一个 SQL 子句——GROUP BY。

13.2.1　语法

我们仔细看看 GROUP BY 子句。图 13-1 是包含 GROUP BY 子句的 SELECT 语句的基本语法图。

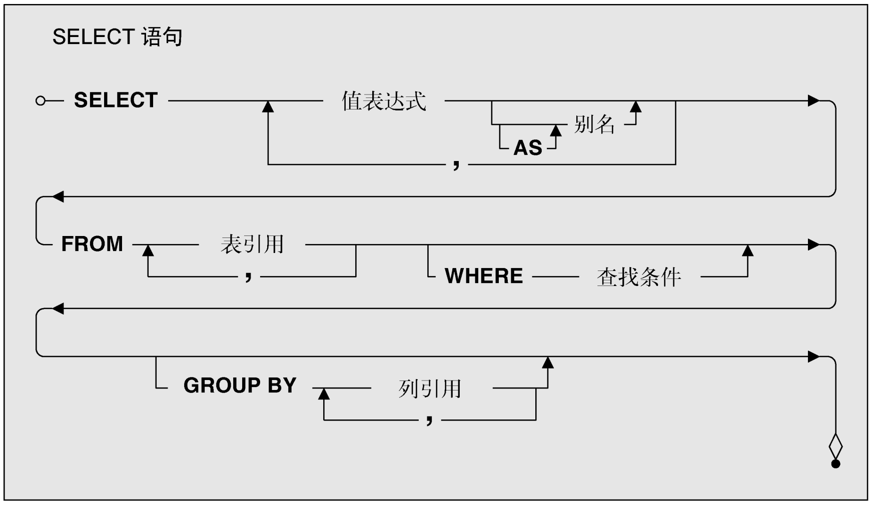


图 13-1　包含 GROUP BY 子句的 SELECT 语句的语法图

本书前面说过，可使用 FROM 子句来指定充当数据源的表。FROM 子句可简单到只包含一个表名，也可复杂到连接多个表。第 8 章讨论过，甚至可将嵌入的表子查询（SELECT 语句）作为表引用。接下来，可使用 WHERE 子句从 FROM 子句提供的结果集中提取或排除特定的行。WHERE 子句在第 6 章详细介绍过。

当你使用 GROUP BY 子句时，数据库系统返回的结果集看起来像是根据该子句中指定的列进行了排序。这是因为有些优化器首先会在内部对数据排序，以提高 GROUP BY 的处理速度。别忘了，如果要按特定的顺序排列，必须使用 ORDER BY 子句。

❖ **说明**　通过添加 GROUP BY 子句，可告诉数据库系统，你要根据 FROM 和 WHERE 子句返回的逻辑表中的哪些列来将行分组。所有指定列的值都相同的行将分成一组。在 GROUP BY 子句中，可指定通过 SELECT 子句的值表达式指定的任何列，还可使用前一章讨论的任何聚合函数对每组执行计算。

下面来将 GROUP BY 子句付诸应用，看看如何按演唱组合计算有关合约价格的信息——这个示例在本章前面提到过。图 13-2 列出了为解决这个问题需要用到的表。

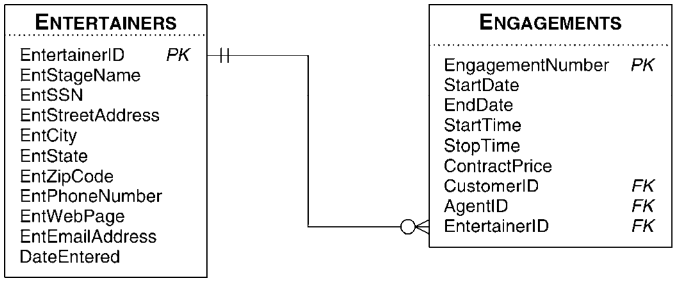


图 13-2 Entertainers 表和 Engagements 表之间的关系

❖ **说明** 贯穿本章都使用了第 4 章介绍的“请求/转换/整理/SQL”流程。

“将演出合约按演唱组合分组，并显示每组的组名、合约数、合约总价、最低合约价格、最高合约价格和平均合约价格。”

［提示：如果请求指出对于高层级的每个值（这里是每个演唱组合），都需要计算其涉及的所有低层值（这里是合约价格）的个数、总和、最小值、最大值或平均值，就说明需要使用聚合函数并进行分组。别忘了，每个演唱组合都可能有很多演出合约。］

转换	Select entertainer name, the count of contracts, the sum of the contract price, the lowest contract price, the highest contract price, and the average contract price from the entertainers table joined with the engagements table on entertainer ID, grouped by entertainer name（基于演唱组合 ID 相同将演唱组合表内连接到演出合约表并按艺名分组，再选择艺名、合约数、合约价格总和、最低合约价格、最高合约价格和平均合约价格）
整理	Select entertainer name, ~~the~~ count ~~of~~ (*) ~~contracts~~, ~~the~~ sum ~~of the~~ (contract price), ~~the lowest~~ min(contract price), ~~the highest~~ max(contract price), ~~and the average~~ avg(contract price) from ~~the~~ entertainers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on entertainers.entertainer ID ~~in the entertainers table matches~~ = engagements.entertainer ID ~~in the engagements table~~, group~~ed~~ by entertainer name

SQL

```
SELECT Entertainers.EntStageName,
   COUNT(*) AS NumContracts,
   SUM(Engagements.ContractPrice) AS TotPrice,
   MIN(Engagements.ContractPrice) AS MinPrice,
   MAX(Engagements.ContractPrice) AS MaxPrice,
   AVG(Engagements.ContractPrice) AS AvgPrice
FROM Entertainers
INNER JOIN Engagements
ON Entertainers.EntertainerID =
   Engagements.EntertainerID
GROUP BY Entertainers.EntStageName
```

请注意，像前一章介绍的那样，我用 MIN 替换了“最低”，用 MAX 替换了“最高”，用 AVG 替换了“平均”。我还使用了 COUNT(*)，因为我要计算总演出合约数，而不管它是否包含 Null 值。通过添加 GROUP BY 子句，可获得每个演唱组合的聚合计算结果，这也让我能够在 SELECT 子句中包含艺名。我将这个查询保存到了示例数据库中，并将其命名为 CH13_Aggregate_Contract_Info_By_Entertainer。

你是否认为在上述查询返回的结果中，每个演唱组合各占一行呢？对于没有任何演出合约的演唱组合，结果将如何呢？如果你还记得第 9 章介绍的外连接，可能想像下面这样解决这个问题：

SQL

```
SELECT Entertainers.EntStageName,
   COUNT(*) AS NumContracts,
   SUM(Engagements.ContractPrice) AS TotPrice,
   MIN(Engagements.ContractPrice) AS MinPrice,
   MAX(Engagements.ContractPrice) AS MaxPrice,
   AVG(Engagements.ContractPrice) AS AvgPrice
FROM Entertainers
LEFT OUTER JOIN Engagements
ON Entertainers.EntertainerID =
   Engagements.EntertainerID
GROUP BY Entertainers.EntStageName
```

有趣的是，所有聚合函数都忽略包含 Null 值的行。对于没有任何演出合约的演唱组合，上述查询返回的 TotPrice、MinPrice、MaxPrice 和 AvgPrice 列都为空值（Null），但返回的 NumContracts 值为 1。怎么会这样呢？这条 SQL 语句使用的是 COUNT(*)，而它不忽略任何返回的行。对于没有任何演出合约的演唱组合，外连接返回一行，因此行数 1 是正确的（我将这个查询保存到了示例数据库中，并将其命名为 CH13_Aggregate_Contract_Info_All_Entertainers_WRONG）。然而，如果你还记得前一章介绍的内容，就知道也可使用 COUNT(值表达式)，让数据库系统只考虑指定值表达式或列不为 Null 的行。下面来微调这个查询。

SQL

```
SELECT Entertainers.EntStageName,
   COUNT(Engagements.EntertainerID) AS NumContracts,
   SUM(Engagements.ContractPrice) AS TotPrice,
   MIN(Engagements.ContractPrice) AS MinPrice,
   MAX(Engagements.ContractPrice) AS MaxPrice,
   AVG(Engagements.ContractPrice) AS AvgPrice
FROM Entertainers
LEFT OUTER JOIN Engagements
ON Entertainers.EntertainerID =
   Engagements.EntertainerID
GROUP BY Entertainers.EntStageName
```

对于没有任何演出合约的演唱组合，Engagements 表中的 EntertainerID 列为 Null，因此不会考虑这样的行。如果你运行这个查询，将看到对于没有任何演出合约的演唱组合，NumContracts 为正确值 0（我将这个查询保存到了示例数据库中，并将其命名为 CH13_Aggregate_Contract_Info_All_Entertainers）。

如果要根据多种值进行分组，该怎么办呢？我们再来看看前面的问题，但从顾客而不是演唱组合的角度看，并假设要在结果集中显示顾客的姓和名。图 13-3 显示了需要用到的表。

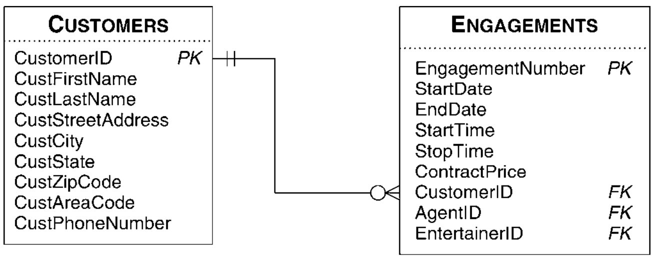


图 13-3　Customers 表和 Engagements 表之间的关系

“列出每位顾客的名、姓、合约数、合约价格总和、最低合约价格、最高合约价格和平均合约价格。”

转换

Select customer last name, customer first name, the count of contracts, the sum of the contract price, the lowest contract price, the highest contract price, and the average contract price from the customers table joined with the engagements table on customer ID, grouped by customer last name and customer first name（基于顾客 ID 相同将顾客表内连接到演出合约表，并按顾客姓名分组，再选择顾客的姓、名、合约数、合约价格总和、最高合约价格、最低合约价格和平均合约价格）

整理

Select cust~~omer~~ last name, cust~~omer~~ first name, ~~the~~ count ~~of~~ (*) ~~contracts, the~~ sum ~~of the~~ (contract price), ~~the lowest~~ min(contract price), ~~the highest~~ max(contract price), ~~and the average~~ avg(contract price) from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on customers.customer ID ~~in the customers table matches~~ = engagements.customer ID ~~in the engagements table~~, group~~ed~~ by customer last name, ~~and~~ customer first name

SQL

```
SELECT Customers.CustLastName,
   Customers.CustFirstName,
   COUNT(*) AS NumContracts,
   SUM(Engagements.ContractPrice) AS TotPrice,
   MIN(Engagements.ContractPrice) AS MinPrice,
   MAX(Engagements.ContractPrice) AS MaxPrice,
   AVG(Engagements.ContractPrice) AS AvgPrice
FROM Customers
INNER JOIN Engagements
ON Customers.CustomerID =
   Engagements.CustomerID
GROUP BY Customers.CustLastName,
   Customers.CustFirstName
```

结果类似于下面这样(我将这个查询保存到了示例数据库 Entertainment Agency 中,并将其命名为 CH13_Aggregate_Contract_Info_By_Customer)。

CustLast Name	CustFirst Name	Num Contracts	TotPrice	MinPrice	MaxPrice	AvgPrice
Berg	Matt	9	$13,170.00	$200.00	$2,675.00	$1,463.33
Brehm	Peter	7	$7,250.00	$290.00	$3,800.00	$1,035.71
Ehrlich	Zachary	13	$12,455.00	$230.00	$1,550.00	$958.08
Hallmark	Elizabeth	8	$25,585.00	$410.00	$14,105.00	$3,198.13
Hartwig	Doris	8	$10,795.00	$140.00	$2,750.00	$1,349.38
Keyser	Liz	7	$4,685.00	$200.00	$1,490.00	$669.29
McCrae	Dean	11	$11,800.00	$290.00	$2,570.00	$1,072.73
Patterson	Kerry	7	$6,815.00	$110.00	$2,930.00	$973.57
<< 其他行 >>						

要显示顾客姓名，需要使用两列，因此我必须在 GROUP BY 子句中包含它们。别忘了，要在输出中显示不是聚合计算结果的列，必须在 GROUP BY 子句中包含它。我没有在 GROUP BY 子句中包含 ContractPrice 列，因为我在很多聚合函数表达式中使用了它。如果在 GROUP BY 子句中包含这列，结果将是每位顾客的每种合约价格一组，而 MIN、MAX 和 AVG 都将返回根据价格分组计算得到的值，而仅当给定顾客有多个价格相同的合约时，COUNT 返回的值才大于 1。只要想一想就知道，要找出有多份同价合约的顾客，根据顾客和合约价格分组并使用 COUNT 是一种不错的方式。

你是否认为这个查询的结果中包含没有签订任何演出合约的顾客呢？如果你的答案是“不包含”，那么回答正确！要获取有关每位顾客的数据，而不管他是否签订了演出合约，必须使用外连接，并根据 Engagements 表中某列的值来计算行数。解决方案与前面讨论的有关演唱组合和演出合约的问题类似。

13.2.2 混合使用列和表达式

假设要在一个输出列中列出顾客姓名，在另一个输出列中列出完整的顾客地址，同时在输出中列出最后的演出日期以及演出合约价格总和。顾客姓名包含在两列（CustFirstName 和 CustLastName）中；而要获得完整的顾客地址，需要用到 CustStreetAddress、CustCity、CustState 和 CustZipCode 列。下面来看看如何编写解决这个问题的 SQL（我将这个查询保存到了示例数据库 Entertainment Agency 中，并将其命名为 CH13_Customers_Last_Booking）。

“列出每位顾客的姓名、完整地址、签订的最后一个演出合约的演出日期以及所有合约的总价。”

转换	Select customer last name and customer first name as CustomerFullName; street address, city, state, and ZIP Code as CustomerFullAddress; the latest contract start date; and the sum of the contract price from the customers table joined with the engagements table on customer ID, grouped by customer last name, customer first name, customer street address, customer city, customer state, and customer ZIP Code（基于顾客 ID 相同将顾客表内连接到演出合约表，并根据顾客的姓、名、街道地址、城市、州和邮政编码分组，再选择姓和名作为 CustomerFullName，选择街道地址、城市、州和邮政编码作为 CustomerFullAddress，再选择签约的最后一次演出的开始日期和合约价格总和）
整理	Select cust~~omer~~ last name ~~and~~ || ', ' || cust~~omer~~ first name as CustomerFullName, street address~~,~~ || ', ' || city~~,~~ || ', ' || state~~, and~~ || ' ' || ZIP Code as CustomerFullAddress, ~~the latest~~ max(contract start date) as latest date, ~~and the~~ sum ~~of the~~ (contract price) as total contract price from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on customers.customer ID ~~in the customers table matches~~ = engagements.customer ID ~~in the engagements table~~ group~~ed~~ by cust~~omer~~ last name, cust~~omer~~ first name, cust~~omer~~ street address, cust~~omer~~ city, cust~~omer~~ state, ~~and~~ cust~~omer~~ ZIP Code

SQL

```
SELECT Customers.CustLastName || ', ' ||
   Customers.CustFirstName AS CustomerFullName,
   Customers.CustStreetAddress || ', ' ||
   Customers.CustCity || ', ' ||
   Customers.CustState || ' ' ||
   Customers.CustZipCode AS CustomerFullAddress,
   MAX(Engagements.StartDate) AS LatestDate,
   SUM(Engagements.ContractPrice),
   AS TotalContractPrice
```

```
FROM Customers
INNER JOIN Engagements
ON Customers.CustomerID =
   Engagements.CustomerID
GROUP BY Customers.CustLastName,
   Customers.CustFirstName,
   Customers.CustStreetAddress,
   Customers.CustCity, Customers.CustState,
   Customers.CustZipCode
```

请注意，在 GROUP BY 子句中，必须列出在不包含聚合函数的输出表达式中使用的每列。StartDate 和 ContractPrice 是在聚合表达式中使用的，因此不需要在 GROUP BY 子句中列出它们。实际上，根据 StartDate 或 ContractPrice 进行分组不合逻辑，因为我要使用它们基于特定的顾客执行聚合计算。例如，如果根据 StartDate 进行分组，MAX(StartDate）将返回分组时基于的值，而表达式 SUM(ContractPrice)将返回给定顾客在给定日期的合约价格总和，这不太可能得到多个合约的价格总和，除非给定顾客有多个给定日期的演出合约。

13.2.3　在 WHERE 子句中的子查询中使用 GROUP BY

第 11 章介绍了聚合函数 COUNT 和 MAX，演示了如何根据使用子查询取回的聚合值来筛选行。第 12 章介绍了如何在子查询筛选器中使用 MIN、AVG 和 SUM。下面来看一个例子，它要求在一个子查询中使用聚合函数和 GROUP BY 子句。

“显示这样的演出合约的合约价格，即它比其他任何顾客的所有合约总价都高。”

转换　Select customer first name, customer last name, engagement start date, and engagement contract price from the customers table joined with the engagements table on customer ID where the contract price is greater than the sum of all contract prices from the engagements table for customers other than the current customer, grouped by customer ID（基于顾客 ID 相同将顾客表内连接到演出合约表，对得到的结果集中的每行做如下处理：从该结果集中选择顾客 ID 与当前顾客 ID 不同的行，根据顾客 ID 将它们分组，并计算各组的合约价格总和，再将当前合约价格与这些价格总和比较，如果比它们都大，就选择当前行中的顾客名、顾客姓、演出开始日期和演出合约价格）

整理　Select cust~~omer~~ first name, cust~~omer~~ last name, engagement start date, ~~and~~ engagement contract price from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on customers.customer ID ~~in the customers table matches~~ = engagements.customer ID ~~in the engagements table~~ where ~~the~~ contract price ~~is greater than~~ > ALL (select ~~the~~ sum ~~of all~~ (contract price~~s~~) from ~~the~~ engagements ~~table~~ as E2 ~~for~~ where E2.customers ID <> ~~other than the current~~ customers.customer ID~~,~~ group~~ed~~ by E2.customer ID)

SQL

```
SELECT Customers.CustFirstName,
   Customers.CustLastName,
   Engagements.StartDate,
   Engagements.ContractPrice
FROM Customers
INNER JOIN Engagements
ON Customers.CustomerID =
   Engagements.CustomerID
WHERE Engagements.ContractPrice > ALL
   (Select SUM(ContractPrice)
   FROM Engagements AS E2
   WHERE E2.CustomerID <> Customers.CustomerID
   GROUP BY E2.CustomerID)
```

我们来分析一下其中的子查询。对于主查询将 Customers 内连接到 Engagements 得到的结果集中表示演出合约的每行，这个子查询都计算其他每位顾客的所有合约的价格总和。由于数据库中有多位顾客，因此这个子查询将返回多个总和——其他每位顾客一个。有鉴于此，不能使用简单的大于比较（>），但第 11 章介绍过，要与一组值比较，可使用限定的大于所有比较（> ALL）。如果你在示例数据库 Entertainment Agency 中运行这个查询（我将它保存成了 CH13_Biggest_Big_Contract），将发现只有一个合约满足条件，如下所示：

CustFirstName	CustLastName	StartDate	ContractPrice
Elizabeth	Hallmark	2018-01-22	$14,105.00

13.2.4　模拟 SELECT DISTINCT 语句

在 SELECT 子句中，可使用不包含聚合函数的 GROUP BY 子句，你想到了吗？当你这样做时，效果与使用第 4 章介绍的关键字 DISTINCT 相同（请参阅 4.5 节）。

我们来看一个要获取不同值的简单问题。我将使用这两种方法来解决它。

“列出顾客表中不同的城市名。”

第 1 次转换	Select the unique city names from the customers table（从顾客表中选择不同的城市名）
整理	Select ~~the unique~~ distinct city ~~names~~ from ~~the~~ customers ~~table~~
SQL	`SELECT DISTINCT Customers.CustCity` `FROM Customers`
第 2 次转换	Select city name from the customers table, grouped by city name（从顾客表中选择城市名，并按城市名分组）
整理	Select city ~~name~~ from ~~the~~ customers ~~table,~~ group~~ed~~ by city name

SQL	`SELECT Customers.CustCity`
	`FROM Customers`
	`GROUP BY Customers.CustCityName`

别忘了，GROUP BY 根据指定的分组列将行分组，并为每组返回一行。这种稍微不同的方式得到的结果与关键字 DISTINCT 相同。哪种方式更好呢？在我看来，如果只是想返回不同的行，使用关键 DISTINCT 的语句更清晰；但使用 GROUP BY 时，运行速度更快。另外，使用 GROUP BY 可获得有关数据的更多信息。请看下面的查询：

SQL

```
SELECT Customers.CustCity, Count(*) as
   CustPerCity
FROM Customers
GROUP BY Customers.CustCityName
```

这个查询不仅取回不同的城市名，还指出了各个城市都有多少顾客。很酷吧？

13.3　一些限制

前面说过，添加 GROUP BY 子句后，将给查询带来一些限制。我们来说说这些限制，以免你落入常见的陷阱。

13.3.1　列方面的限制

添加 GROUP BY 子句意味着让数据库系统将 WHERE 子句从 FROM 子句指定的表中筛选出来的行进行分组。在 SELECT 子句中，你想使用多少个聚合表达式都可以。在这些表达式中，可使用 FROM 和 WHERE 子句定义的表中的任何列，但我在前面的一个示例中指出过，在聚合表达式中用于分组的列可能没有意义。

在 SELECT 子句中，如果有表达式引用了列但不包含聚合函数，就必须在 GROUP BY 子句中列出这些列。一种常见的错误是，在非聚合表达式中可引用任何列，只要它包含在独特的行中。我们来看一个错误的查询，它使用了一个主键值，而我知道，主键值根据定义是独一无二的。

“显示顾客的 ID、姓名和所有演出合约的合约价格总和。”

转换	Select customer ID, customer first name, and customer last name as CustFullName, and the sum of contract prices as TotalPrice from the customers table joined with the engagements table on customer ID, grouped by customer ID［基于顾客 ID 相同将顾客表内连接到演出合约表，根据顾客 ID 分组，并选择顾客的 ID、姓和名（作为 CustFullName）和合约价格总和（作为 TotalPrice）］
整理	Select customer ID, cust~~omer~~ first name ~~and~~ || ‘ ’ || cust~~omer~~ last name as CustFullName, ~~and the~~ sum ~~of~~ (contract price)~~s~~ as TotalPrice from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on customers.customer ID ~~in the customers table matches~~ = engagements.customer ID ~~in the engagements table~~, group~~ed~~ by customer ID

```
SQL         SELECT Customers.CustomerID,
               Customers.CustFirstName || ' ' ||
               Customers.CustLastName AS CustFullName,
               SUM(Engagements.ContractPrice) AS TotalPrice
            FROM Customers
            INNER JOIN Engagements
            ON Customers.CustomerID =
               Engagements.CustomerID
            GROUP BY Customers.CustomerID
```

我知道，每位顾客的 CustomerID 都是独一无二的，仅根据 CustomerID 进行分组就应该足以取回不同的顾客姓和名。然而，SQL 是一种基于语法而不是语义的语言，换而言之，SQL 不会考虑数据库表的设计隐含的信息，包括列是否是主键。SQL 要求查询必须从语法上说是“纯粹”的，无须对低层表的设计有任何了解就能翻译。因此，在完全遵循 SQL 标准的数据库系统中，上述 SQL 语句将不能执行，因为其中的 SELECT 子句使用了 CustFirstName 和 CustLastName 列，而这两个列不是在聚合函数中使用的，同时又没有包含在 GROUP BY 子句中。令人惊讶的是，像上面这样的查询在 MySQL 和 PostgreSQL 中能够运行。正确的 SQL 查询如下：

```
SQL         SELECT Customers.CustomerID,
               Customers.CustFirstName || ' ' ||
               Customers.CustLastName AS CustFullName,
               SUM(Engagements.ContractPrice) AS TotalPrice
            FROM Customers
            INNER JOIN Engagements
            ON Customers.CustomerID =
               Engagements.CustomerID
            GROUP BY Customers.CustomerID,
               Customers.CustFirstName,
               Customers.CustLastName
```

可能看起来有点小题大作，但这才是正确的方式！

❖ **说明**　在有些数据库系统中，在 SELECT 子句中使用的表达式必须与 GROUP BY 子句中完全相同，Oracle 和 Microsoft Office Access 就是支持这样做或要求必须这样做的例子。（Microsoft Office Access 允许你采用任意一种方式）。例如，在刚才的示例中，在 GROUP BY 子句中不能分别列出各列，而必须像下面这样做：

```
GROUP BY Customers.CustomerID,
   Customers.CustFirstName || ' ' ||
   Customers.CustLastName
```

这不符合 SQL 标准，但在你使用的数据库系统中，可能必须这样做。

13.3.2　根据表达式分组

前面通过一些示例演示了编写不包含聚合函数的表达式的正确方式。一种常见的错误是，试图根据 SELECT 子句中使用的表达式（而不是列）进行分组。请记住，在 GROUP BY 子句中，只能引用 FROM 和 WHERE 子句生成的列，而不能使用 SELECT 子句中使用的表达式。

为了说明这意味着什么，我们再来看一下前面解决的一个问题，但故意犯刚才所说的错误（这里将省略转换和整理，因为前面做过）。

“显示居住在华盛顿州的每位顾客的姓名、完整地址、签约的最后一次演出的日期以及所有合约的总价。”

```
SQL         SELECT Customers.CustLastName || ', ' ||
               Customers.CustFirstName AS CustomerFullName,
               Customers.CustStreetAddress || ', ' ||
               Customers.CustCity || ', ' ||
               Customers.CustState || ' ' ||
               Customers.CustZipCode AS CustomerFullAddress,
               MAX(Engagements.StartDate) AS LatestDate,
```

```
      SUM(Engagements.ContractPrice)
      AS TotalContractPrice
   FROM Customers
   INNER JOIN Engagements
   ON Customers.CustomerID =
      Engagements.CustomerID
   WHERE Customers.CustState = 'WA'
   GROUP BY CustomerFullName,
      CustomerFullAddress
```

在有些数据库系统中，这个查询能够运行，但它是错误的。数据库系统执行 FROM、WHERE 和 GROUP BY 子句时，根本就不存在 CustomerFullName 和 CustomerFullAddress 列。在 FROM 和 WHERE 子句生成的结果中，GROUP BY 子句找不到这些列，因此在严格遵循 SQL 标准的数据库系统中，这个查询将导致语法错误。

前面介绍了一种解决这个问题的正确方式：必须列出在表达式 CustomerFullName 和 CustomerFullAddress 中使用的所有列。另一种方式是，在 FROM 子句中嵌入表查询来生成计算列，如下所示：

```
SQL        SELECT CE.CustomerFullName,
              CE.CustomerFullAddress,
              MAX(CE.StartDate) AS LatestDate,
              SUM(CE.ContractPrice) AS TotalContractPrice
           FROM
               (SELECT Customers.CustLastName || ', ' ||
              Customers.CustFirstName AS CustomerFullName,
                 Customers.CustStreetAddress || ', ' ||
                 Customers.CustCity || ', ' ||
                 Customers.CustState || ' ' ||
                 Customers.CustZipCode AS CustomerFullAddress,
                 Engagements.StartDate,
                 Engagements.ContractPrice
              FROM Customers
              INNER JOIN Engagements
              ON Customers.CustomerID =
                 Engagements.CustomerID
              WHERE Customers.CustState = 'WA')
           AS CE
           GROUP BY CE.CustomerFullName,
              CE.CustomerFullAddress
```

这之所以可行，是因为在 FROM 子句中生成了输出列 CustomerFullName 和 CustomerFullAddress。但必须承认，这让查询非常复杂。实际上，相比于在 FROM 子句中使用表达式来生成列，在 GROUP BY 子句中列出要在非聚合表达式中使用的所有原始列是更佳的选择。我将这个查询保存到了示例数据库 Entertainment Agency 中，并将其命名为 CH13_Customers_Total_Contract。

13.4　GROUP BY 的用途

至此，对于如何使用聚合函数和 GROUP BY 子句来获取分组小计，你应该有了相当深入的认识。为了让你知道 GROUP BY 的用途有多广泛，最佳的方式是列举一些使用这个子句可解决的问题（有关该子句的使用示例，请参阅 13.5 节）。

“列出每家供应商及其平均可在多少天内将商品送达。”
“显示每种商品的名称和总销量。”
“列出每位顾客的姓名、下单日期及每个下单日期订购的商品总价。”
“显示每个演唱组合的 ID、成员和平均报酬（演唱组合的合约价格总和除以成员数量）。”
“显示每个经纪人的姓名及其签订的演出合约的总价和得到的总佣金。”
“列出每种课程类别的名称以及选修了该类别中已结束课程的每位学生的姓名和平均成绩。”
“显示每种课程类别的名称及其包含的课程数量。”
“列出每位教员及其讲授的课程数量。”
“显示每次联赛的 ID 和比赛地点，及其包含的比赛场次的 ID、各个参赛球队的名称和总让分得分。”

“显示每个投球手的姓名和平均原始得分。”

“显示每种食材类型及其包含的食材被用于制作多少种菜品。”

“如果要制作我的烹饪书中列出的所有菜品，需要每种食材各多少？”

13.5　语句举例

知道使用 GROUP BY 子句编写查询的机制以及可使用它来解决的一些问题类型后，下面来看一组示例，它们都要求对信息进行分组，且都来自示例数据库。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库中，名称以 CH13 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明**　别忘了，这些示例中使用的所有列名和表名都来自示例数据库，有关这些数据库的结构，请参阅附录 B。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。这些示例都假定你深入研究并掌握了本书前面介绍的概念，尤其是连接和子查询。

- **Sales Orders 数据库**

“列出每位顾客的姓名、下单日期及每个下单日期订购的商品总价。”

转换/整理	Select cust~~omer~~ first name ~~and~~ || ‘ ‘ || cust~~omer~~ last name as CustFullName, order date, ~~and the~~ sum ~~of~~ (quoted price ~~times~~ * quantity ordered) as TotalCost from ~~the~~ customers ~~table~~ inner join~~ed with the~~ orders ~~table~~ on customers.customer ID ~~in the customers table matches~~ = orders.customer ID ~~in the orders table~~, ~~and then inner joined with the~~ order details ~~table~~ on orders.order number ~~in the orders table matches~~ = order_details.order number ~~in the order details table~~, group~~ed~~ by customer first name, customer last name, ~~and~~ order date［依次基于顾客 ID 相同将顾客表内连接到订单表、基于订单号相同内连接到订单详情表，根据顾客的名和姓以及订单日期分组，选择顾客的名和姓（作为 CustFullName）、订单日期以及报价和订购数量乘积的总和（作为 TotalCost）］

SQL

```
SELECT Customers.CustFirstName || ' ' ||
   Customers.CustLastName AS CustFullName,
   Orders.OrderDate,
   SUM(Order_Details.QuotedPrice *
   Order_Details.QuantityOrdered) AS TotalCost
FROM (Customers
INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber =
   Order_Details.OrderNumber
GROUP BY Customers.CustFirstName,
   Customers.CustLastName, Orders.OrderDate
```

CH13_Order_Totals_By_Customer_And_Date（847 行）

CustFullName	OrderDate	TotalCost
Alaina Hallmark	2017-09-03	$4,699.98
Alaina Hallmark	2017-09-15	$4,433.95
Alaina Hallmark	2017-09-22	$353.25
Alaina Hallmark	2017-09-23	$3,951.90
Alaina Hallmark	2017-10-01	$10,388.68
Alaina Hallmark	2017-10-13	$3,088.00
Alaina Hallmark	2017-10-23	$6,775.06
Alaina Hallmark	2017-10-31	$15,781.10
	<< 其他行 >>	

- **Entertainment Agency 数据库**

“显示每个演唱组合的 ID、成员和平均报酬（演唱组合的合约价格总和除以成员数量）。”

❖ **说明** 这个问题确实很棘手，因为同一位艺人可能属于多个演唱组合。必须计算每个演唱组合的合约价格总和，再除以成员数（假设每个成员的报酬相同）。要获取成员数，需要一个根据演唱组合 ID（不是成员 ID）进行筛选的子查询，这意味着也必须根据演唱组合 ID 进行分组。哦，对了，别忘了将已退出（Status = 3）的成员排除在外。

转换/整理

Select entertainer ID, ~~member~~ first name, ~~member~~ last name, ~~and the~~ sum ~~of~~(contract price)~~s divided by~~ / ~~the~~ (select count(*) ~~of active members~~ from entertainer members as EM2 ~~in the current entertainer group~~ where status ~~is not equal to~~ <> ~~not active~~ 3 and ~~the~~ EM2 ~~table~~ entertainer ID ~~equals~~ = ~~the~~ entertainer members ~~table~~ entertainer ID) from ~~the~~ members ~~table~~ inner join~~ed with the~~ entertainer members ~~table~~ on members.member ID ~~in the members table matches~~ = entertainer_members.member ID ~~in the entertainer members table, then~~ inner join~~ed with the~~ entertainers ~~table~~ on entertainers.entertainer ID ~~in the entertainers table matches~~ = entertainer_members.entertainer ID ~~in the entertainer members table~~, ~~and finally~~ inner join~~ed with the~~ engagements ~~table~~ on entertainers.entertainer ID ~~in the entertainers table matches~~ = engagements.entertainer ID ~~in the engagements table,~~ where member status ~~is not equal to~~ <> ~~not active 3,~~ group~~ed~~ by entertainer ID, m~~em~~ber first name, ~~and member~~ last name, ~~sorted~~ order by m~~em~~ber last name［依次基于成员 ID 相同将成员表内连接到演唱组合成员表、基于演唱组合 ID 相同内连接到演唱组合表、基于演唱组合 ID 相同内连接到演出合约表，选择成员状态不为“已退出”（3）的行，并根据演唱组合 ID、成员的名和姓将这些行分组；对于每个分组，计算合约价格总和，再从演唱组合成员表（别名为 EM2）中选择演唱组合 ID 与当前分组相同且成员状态不为“已退出”（3）的行，使用 count(*)计算行数，并将合约价格总和除以这个行数，得到平均报酬；选择每个分组的演唱组合 ID、成员姓和名以及平均报酬，并按成员姓排序］

SQL

```
SELECT Entertainers.EntertainerID,
   Members.MbrFirstName, Members.MbrLastName,
   SUM(Engagements.ContractPrice)/
      (SELECT COUNT(*)
      FROM Entertainer_Members AS EM2
      WHERE EM2.Status <> 3
      AND EM2.EntertainerID =
      Entertainers.EntertainerID)
   AS MemberPay
FROM ((Members
INNER JOIN Entertainer_Members
ON Members.MemberID =
   Entertainer_Members.MemberID)
INNER JOIN Entertainers
ON Entertainers.EntertainerID =
   Entertainer_Members.EntertainerID)
INNER JOIN Engagements
ON Entertainers.EntertainerID =
   Engagements.EntertainerID
WHERE Entertainer_Members.Status <> 3
GROUP BY Entertainers.EntertainerID,
   Members.MbrFirstName, Members.MbrLastName
ORDER BY Members.MbrLastName
```

CH13_Member_Pay（39 行）

EntertainerID	MbrFirstName	MbrLastName	MemberPay
1010	Kendra	Bonnicksen	$2,887.50
1013	Kendra	Bonnicksen	$3,767.50
1007	Robert	Brown	$2,975.00
1008	Robert	Brown	$6,816.00
1008	George	Chavez	$6,816.00
1013	George	Chavez	$3,767.50
1010	Caroline	Coie	$2,887.50
1013	Caroline	Coie	$3,767.50
<< 其他行 >>			

- **School Scheduling 数据库**

"列出每种课程类别的名称以及选修了该类别中已结束课程的每位学生的姓名和平均成绩。"

转换/整理

Select category description, stud~~ent~~ first name, stud~~ent~~ last name, ~~and the average~~ AVG(~~of~~ grade) as AvgOfGrade from ~~the~~ categories ~~table~~ inner join~~ed with the~~ subjects ~~table~~ on categories.category ID ~~in the categories table matches~~ = subjects.category ID ~~in the subjects table, then~~ inner join~~ed with the~~ classes ~~table~~ on subjects.subject ID ~~in the subjects table matches~~ = classes.subject ID ~~in the classes table~~, ~~then~~ inner join~~ed with the~~ student schedules ~~table~~ on classes.class ID ~~in the classes table matches~~ = student_schedules.class ID ~~in the student schedules table~~, ~~then~~ inner join~~ed with the~~ student class status ~~table~~ on student_class_status.class status ~~in the student class status table matches~~ = student_schedules.class status ~~in the student schedules table, and finally~~ inner join~~ed with the~~ students ~~table~~ on students.student ID ~~in the students table matches~~ = student_schedules.student ID ~~in the student schedules table~~ where class status description ~~is~~ = 'Completed,' group~~ed~~ by category description, student first name, ~~and~~ student last name
［依次基于类别 ID 相同将类别表内连接到科目表、基于科目 ID 相同内连接到课程表、基于课程 ID 相同内连接到学生选课表、基于课程状态相同内连接到学生课程表、基于学生 ID 相同内连接到学生表；选择课程状态描述为 Completed 的行，并根据类别描述以及学生的名和姓分组；选择类别描述、学生的名和姓以及使用 AVG 计算得到的平均成绩（作为 AvgOfGrade）］

SQL

```
SELECT Categories.CategoryDescription,
   Students.StudFirstName,
   Students.StudLastName,
   AVG(Student_Schedules.Grade) AS AvgOfGrade
FROM ((((Categories
INNER JOIN Subjects
ON Categories.CategoryID = Subjects.CategoryID)
INNER JOIN Classes
ON Subjects.SubjectID = Classes.SubjectID)
INNER JOIN Student_Schedules
ON Classes.ClassID = Student_Schedules.ClassID)
INNER JOIN Student_Class_Status
ON Student_Class_Status.ClassStatus =
   Student_Schedules.ClassStatus)
INNER JOIN Students
ON Students.StudentID =
   Student_Schedules.StudentID
WHERE Student_Class_Status.ClassStatusDescription =
   'Completed'
GROUP BY Categories.CategoryDescription,
   Students.StudFirstName,
   Students.StudLastName
```

CH13_Student_GradeAverage_By_Category（63 行）

Category Description	StudFirst Name	StudLast Name	AvgOfGrade
Accounting	Doris	Hartwig	80.51
Accounting	Elizabeth	Hallmark	91.12
Accounting	Kendra	Bonnicksen	88.50
Accounting	Richard	Lum	79.61
Accounting	Sarah	Thompson	77.34
Art	Doris	Hartwig	82.19
Art	George	Chavez	83.63
Art	John	Kennedy	87.65
Art	Kerry	Patterson	99.83
Art	Michael	Viescas	73.37
<< 其他行 >>			

- **Bowling League 数据库**

"显示每次联赛的 ID 和比赛地点，及其包含的比赛场次的 ID、各个参赛球队的名称和总让分得分。"

转换/整理

Select tourney ID, tourney location, match ID, team name, ~~and the~~ sum ~~of~~(handicap score) as TotHandiCapScore from ~~the~~ tournaments ~~table~~ inner join~~ed with the~~ tourney matches ~~table~~ on tournaments.tourney ID ~~in the tournaments table matches~~ = tourney_matches.tourney ID ~~in the tourney matches table, then~~ inner join~~ed with the~~ match games ~~table~~ on tourney_matches.match ID ~~in the tourney matches table matches~~ = match_games.match ID ~~in the match games table, then~~ inner join~~ed with the~~ bowler scores ~~table~~ on match_ games.match ID ~~in the match games table matches~~ = bowler_scores.match ID ~~in the bowler scores table~~ and match_games.game number ~~in the match games table matches~~ = bowler_scores.game number ~~in the bowler scores table~~, ~~then~~ inner join~~ed with the~~ bowlers ~~table~~ on bowlers.bowler ID ~~in the bowlers table matches~~ = bowler_scores.bowler ID ~~in the bowler scores table, and finally~~ inner join~~ed with the~~ teams ~~table~~ on teams.team ID ~~in the teams table matches~~ = bowlers.team ID ~~in the bowlers table~~, group~~ed~~ by tourney ID, tourney location, match ID, ~~and~~ team name[依次基于联赛 ID 相同将联赛表内连接到联赛场次表、基于场次 ID 相同内连接到场次局次表、基于场次 ID 相同和局次编号相同内连接到投球手得分表、基于投球手 ID 相同内连接到投球手表、基于球队 ID 相同内连接到球队表；根据联赛 ID、联赛地点、场次 ID 和球队 ID 分组；选择联赛 ID、联赛地点、场次 ID、球队名称和总让分得分（作为 TotHandiCapScore）]

SQL

```
SELECT Tournaments.TourneyID,
   Tournaments.TourneyLocation,
   Tourney_Matches.MatchID, Teams.TeamName,
   SUM(Bowler_Scores.HandicapScore)
   AS TotHandiCapScore
FROM ((((Tournaments
INNER JOIN Tourney_Matches
ON Tournaments.TourneyID =
   Tourney_Matches.TourneyID)
INNER JOIN Match_Games
ON Tourney_Matches.MatchID =
   Match_Games.MatchID)
INNER JOIN Bowler_Scores
ON (Match_Games.MatchID =
   Bowler_Scores.MatchID) AND
   (Match_Games.GameNumber =
   Bowler_Scores.GameNumber))
INNER JOIN Bowlers
ON Bowlers.BowlerID = Bowler_Scores.BowlerID)
INNER JOIN Teams
ON Teams.TeamID = Bowlers.TeamID
GROUP BY Tournaments.TourneyID,
   Tournaments.TourneyLocation,
   Tourney_Matches.MatchID, Teams.TeamName
```

如你所见，这个查询的难点在于编写复杂的 JOIN 子句，以正确的方式将所有的表连接起来。

CH13_Tournament_Match_Team_Results（112 行）

Tourney ID	Tourney Location	MatchID	MatchID TeamName	TotHandi CapScore
1	Red Rooster Lanes	1	Marlins	2351
1	Red Rooster Lanes	1	Sharks	2348
1	Red Rooster Lanes	2	Barracudas	2289
1	Red Rooster Lanes	2	Terrapins	2391
1	Red Rooster Lanes	3	Dolphins	2389
1	Red Rooster Lanes	3	Orcas	2395
1	Red Rooster Lanes	4	Manatees	2292
1	Red Rooster Lanes	4	Swordfish	2353
2	Thunderbird Lanes	5	Marlins	2297
2	Thunderbird Lanes	5	Terrapins	2279
		<< 其他行 >>		

"显示每位投球手的最高原始得分。"

转换/整理

Select bowler first name, bowler last name, ~~and the~~ max~~imum~~ (raw score) as HighScore from ~~the~~ bowlers ~~table~~ inner join~~ed with the~~ bowler scores ~~table~~ on bowlers.bowler ID ~~in the bowlers table matches~~ = bowler_scores.bowler ID ~~in the bowler scores table~~, group~~ed~~ by bowler first name, ~~and~~ bowler last name［基于投球手 ID 相同将投球手表内连接到投球手得分表，并根据投球手名和姓分组，再选择投球手名和姓以及 max（原始得分，作为 HighScore）］

SQL

```
SELECT Bowlers.BowlerFirstName,
   Bowlers.BowlerLastName,
   MAX(Bowler_Scores.RawScore) AS HighScore
FROM Bowlers
INNER JOIN Bowler_Scores
ON Bowlers.BowlerID = Bowler_Scores.BowlerID
GROUP BY Bowlers.BowlerFirstName,
   Bowlers.BowlerLastName
```

CH13_Bowler_High_Score_Group（32 行）

BowlerFirstName	BowlerLastName	HighScore
Alaina	Hallmark	180
Alastair	Black	164
Angel	Kennedy	194
Ann	Patterson	165
Bailey	Hallmark	164
Barbara	Fournier	164
Caleb	Viescas	193
Carol	Viescas	150
David	Cunningham	180
David	Fournier	178
	<< 其他行 >>	

- **Recipes 数据库**

“显示每种食材类型及其包含的食材被用于制作多少种菜品。”

❖ **说明**　这里的难点在于，对于特定的食材类型，不能计算同一种菜品多次。例如，如果一种菜品包含多种香草类或奶类食材，那么对于每种食材类型，该菜品应只计算一次。看起来该使用 COUNT(DISTINCT 值表达式)了，不是吗？

转换/整理

Select ingredient class description, ~~and the unique~~ count ~~of~~ (distinct recipe ID) as CountOfRecipeID from ~~the~~ ingredient classes ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on ingredient_classes.ingredient class ID ~~in the ingredient classes table matches~~ = ingredients.ingredient class ID ~~in the ingredients table, and then~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~, group~~ed~~ by ingredient class description［依次基于食材类型 ID 相同将食材类型表内连接到食材表、基于食材 ID 相同内连接到菜品食材表，根据食材类型描述分组，再选择食材类型描述以及不同菜品 ID 的数量（作为 CountOfRecipeID）］

SQL

```
SELECT
   Ingredient_Classes.IngredientClassDescription,
   Count(DISTINCT RecipeID) AS CountOfRecipeID
FROM (Ingredient_Classes
INNER JOIN Ingredients
ON Ingredient_Classes.IngredientClassID =
   Ingredients.IngredientClassID)
INNER JOIN Recipe_Ingredients
ON Ingredients.IngredientID =
   Recipe_Ingredients.IngredientID
GROUP BY
   Ingredient_Classes.IngredientClassDescription
```

CH13_IngredientClass_Distinct_Recipe_Count（19 行）

IngredientClassDescription	CountOfRecipeID
Bottle	1
Butter	3
Cheese	2

（续）

IngredientClassDescription	CountOfRecipeID
Chips	1
Condiment	2
Dairy	2
Fruit	1
Grain	2
Herb	1
<< 其他行 >>	

❖ **说明** 由于 Microsoft Access 不支持 COUNT DISTINCT，因此在 Access 示例数据库中，首先在 FROM 子句中使用了一个表子查询来选择不同的 RecipeID 值，再计算得到的行数。

13.6 小结

本章首先阐述了为何要将数据分组以便获得多个小计。在使用一个示例吊起你的胃口后，介绍了如何使用 GROUP BY 子句来解决这个问题以及其他几个问题，还演示了如何混合使用列表达式和聚合函数。

接下来介绍了一个有趣的示例：于在 WHERE 子句中充当筛选器的子查询中使用 GROUP BY。我指出编写查询时，如果使用了 GROUP BY，且没有使用聚合函数，效果将与在 SELECT 子句中使用 DISTINCT 相同；我提醒你编写 GROUP BY 子句时要小心，应只在其中指定列，而不要使用表达式。

我指出了一些常见的陷阱，以此来结束对 GROUP BY 子句的讨论。我指出了 SQL 不考虑列是否是主键，并列举了你在 SELECT 子句中使用列表达式时可能犯的一些常见错误。

本章最后总结了 GROUP BY 很有用的原因，列举了一系列可使用 GROUP BY 来解决的问题，并通过示例演示了如何使用 GROUP BY 子句来编写查询。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

13.7 练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 列出每家供应商及其平均可在多少天内将商品送达。
  　提示：使用聚合函数 AVG 并按供应商分组。
  　解决方案见 CH13_Vendor_Avg_Delivery（10 行）。

  (2) 显示每种商品的名称和总销量。
  　提示：使用 SUM 计算数量与价格乘积的总和，并按商品名分组。
  　解决方案见 CH13_Sales_By_Product（38 行）。

  (3) 列出所有供应商以及每家供应商销售的商品数量。
  　解决方案见 CH13_Vendor_Product_Count_Group（10 行）。

  (4) 使用子查询解决前一个问题（较难）。
  　解决方案见 CH13_Vendor_Product_Count_Subquery（10 行）。

- **Entertainment Agency 数据库**

  (1) 显示每个经纪人的姓名及其签订的演出合约的总价和得到的总佣金。

提示：必须将合约总价乘以提成比例；另外，务必按提成比例分组。

解决方案见 CH13_Agent_Sales_And_Commissions（8 行）。

- **School Scheduling 数据库**

(1) 显示每种课程类别的名称及其包含的课程数量。

提示：使用 COUNT 并按类别名分组。

解决方案见 CH13_Category_Class_Count（15 行）。

(2) 列出每位教员及其讲授的课程数量。

提示：使用 COUNT 并按教员姓名分组。

解决方案见 CH13_Staff_Class_Count（22 行）。

(3) 使用子查询来解决前一个问题（较难）。

解决方案见 CH13_Staff_Class_Count_Subquery（27 行）。

(4) 刚才的子查询解决方案返回的结果集多了 5 行，你能解释其中的原因吗？可修改问题 2 中的查询，使其返回 27 行吗？如果可以，该如何修改？

提示：考虑使用外连接。

- **Bowling League 数据库**

(1) 显示每个投球手的姓名和平均原始得分。

提示：使用聚合函数 AVG 并按投球手姓名分组。

解决方案见 CH13_Bowler_Averages（32 行）。

(2) 计算每位投球手当前的平均原始得分和让分数。

提示：这是一个“友好”的联盟，让分数为 200 减去平均原始得分之后的 90%。务必通过取整将平均原始得分转换为整数，再用 200 与它相减，然后将乘以 90%得到的最终结果取整。SQL 标准没有定义函数 ROUND，但大多数商业数据库系统提供了，详情请参阅你使用的数据库系统的文档。

解决方案见 CH13_Bowler_Average_Handicap（32 行）。

(3) 通过使用子查询显示每位投球手的最高原始得分（较难）。

解决方案见 CH13_Bowler_High_Score_Subquery（32 行）。

- **Recipes 数据库**

(1) 如果要制作我的烹饪书中列出的所有菜品，需要每种食材各多少？

提示：使用 SUM 并按食材名和度量单位描述分组。

解决方案见 CH13_Total_Ingredients_Needed（65 行）。

(2) 列出每种肉类食材及其被用来制作的菜品数量。

解决方案见 CH13_Meat_Ingredient_Recipe_Count_Group（4 行）。

(3) 使用子查询解决前一个问题（较难）。

解决方案见 CH13_Meat_Ingredient_Recipe_Count_Subquery（11 行）。

(4) 刚才的子查询解决方案返回的结果集多了 7 行，你能解释其中的原因吗？可修改问题 2 中的查询，使其返回 11 行吗？如果可以，该如何修改？

提示：考虑使用外连接。

第 14 章 筛选分组数据

“让校长用语法、胡说八道和知识来迷惑他们的大脑；我坚定地认为，一壶好酒会让天才更独具慧眼。”

——奥利弗·哥德史密斯

本章涵盖如下主题：

- ❑ 一种新的筛选方式
- ❑ 在哪里筛选更好
- ❑ HAVING 的用途
- ❑ 语句举例
- ❑ 小结
- ❑ 练习

第 12 章详细介绍了 SQL 标准中定义的所有聚合函数；第 13 章讨论了如何让数据库系统将行集分组并计算各组的聚合值。分组的好处之一是，可显示基于分组列的表达式，以标识各组。

本章介绍汇总和分组“拼图”中的最后一块。将行分组并计算聚合值后，使用谓词根据聚合值进一步筛选结果集常常很有用。你马上就会看到，这需要使用最后一块“拼图”——HAVING 子句。

14.1 一种新的筛选方式

你知道，将信息分组后，便可使用 MIN、MAX、AVG、SUM 或 COUNT 对每组的所有值执行计算。如果要进一步筛选结果集，可对聚合值进行检查。我们来看一个简单的问题：

“列出演奏爵士乐且成员超过 3 人的演唱组合。”

听起来是不是太难了？图 14-1 列出了解决这个问题需要用到的表。

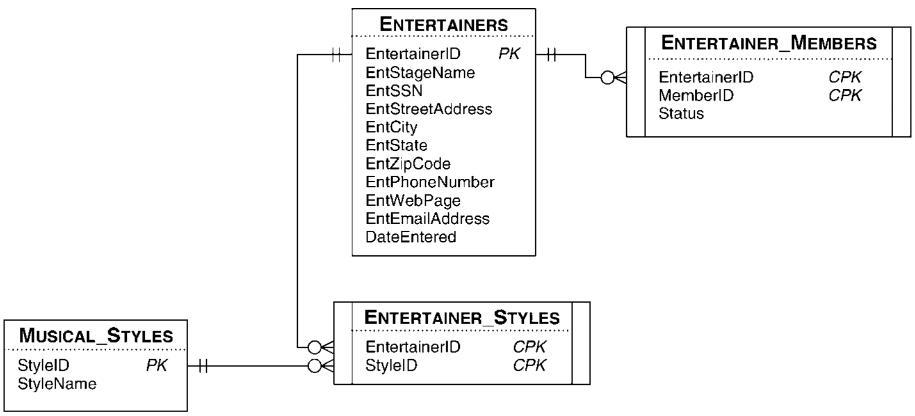


图 14-1　为找出演奏爵士乐且成员超过 3 个的演唱组合需要用到的表

> ❖ **说明**　这里也将使用第 4 章介绍的“请求/转换/整理/SQL”流程。另外，这里还将用到第 8、9 和 11 章介绍的一些连接和子查询知识。

如果不知道 HAVING 子句，你可能会以如下错误的方式解决这个问题。

转换	Select the entertainer stage name and the count of members from the entertainers table joined with the entertainer members table on entertainer ID in the entertainers table matches entertainer ID in the entertainer members table where the entertainer ID is in the selection of entertainer IDs from the entertainer styles table joined with the musical styles table on style ID in the entertainer styles table matches style ID in the musical styles table where the stylename is ‘Jazz’ and where the count of the members is greater than 3, grouped by entertainer stage name（基于风格 ID 相同将演唱组合风格表内连接到音乐风格表，从风格名为 Jazz 的行中选择演唱组合 ID，生成一个演唱组合 ID 列表；基于演唱组合 ID 相同将演唱组合表内连接到演唱组合成员表，选择演唱组合 ID 位于前面生成的演唱组合 ID 列表中且成员超过 3 人的行；根据演唱组合艺名分组，并选择演唱组合艺名和成员数量）
整理	Select ~~the~~ ent~~ertainer~~ stage name ~~and the~~ count(*) ~~of members~~ as CountOfMembers from ~~the~~ entertainers ~~table~~ inner join~~ed with the~~ entertainer members ~~table~~ on entertainers. entertainer ID ~~in the entertainers table matches~~ = entertainer_members.entertainer ID ~~in the entertainer members table~~ where ~~the~~ entertainer ID ~~is~~ in ~~the~~ (selec~~tion of~~ entertainer ID~~s~~ from ~~the~~ entertainer styles ~~table~~ inner join~~ed with the~~ musical styles ~~table~~ on entertainer_styles.style ID ~~in the entertainer styles table matches~~ = musical_styles. style ID ~~in the musical styles table~~ where ~~the~~ style name ~~is~~ = ‘Jazz’) and ~~where the~~ count(*) ~~of the members is greater than~~ > 3~~,~~ group~~ed~~ by entertainer stage name

SQL

```
SELECT Entertainers.EntStageName,
   COUNT(*) AS CountOfMembers
FROM Entertainers
INNER JOIN Entertainer_Members
ON Entertainers.EntertainerID =
   Entertainer_Members.EntertainerID
WHERE Entertainers.EntertainerID
IN
   (SELECT Entertainer_Styles.EntertainerID
   FROM Entertainer_Styles
   INNER JOIN Musical_Styles
   ON Entertainer_Styles.StyleID =
      Musical_Styles.StyleID
   WHERE Musical_Styles.StyleName = 'Jazz')
AND COUNT(*) > 3
GROUP BY Entertainers.EntStageName
```

这条 SQL 语句有何错误呢？关键是在 WHERE 子句（第 6 章介绍过）中引用的列必须位于 FROM 子句定义的表中。COUNT(*)是 FROM 子句生成的列吗？我认为不是。实际上，只有将行分组后，才能计算各组的行数。

看起来需要在 GROUP BY 后面使用一个新子句。图 14-2 说明了包含 HAVING 子句的 SELECT 语句的语法。

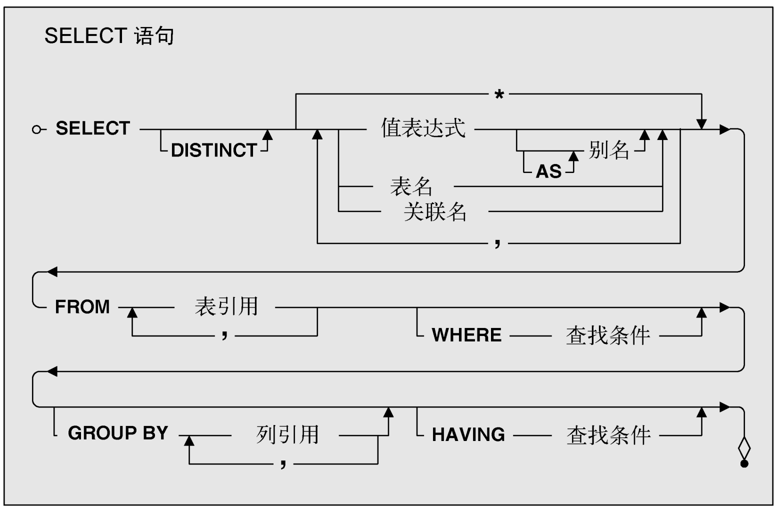


图 14-2　包含所有子句的 SELECT 语句

由于 HAVING 子句在分组后处理行，SQL 标准就可在其查找条件中的谓词中引用的列上做了一些限制。请注意，在 GROUP BY 子句的情况下，HAVING 子句处理 FROM 和 WHERE 子句返回的所有行，就像它们是一组一样。坦率地说，我想不出有什么理由在查询中没有使用 GROUP BY 子句时使用 HAVING 子句。

对于在 HAVING 子句中引用的列，限制与对在分组查询的 SELECT 子句中引用的列一样。在 HAVING 子句的查找条件的谓词中，引用的列必须出现在了 GROUP BY 子句或聚合函数中。这合乎逻辑，因为任何列比较都必须使用对行分组得到的结果：要么是分组值，要么是根据每组计算得到的聚合值。

对 HAVING 有些了解后，我们以正确的方式解决前面的问题：

“列出演奏爵士乐且成员超过 3 人的演唱组合。”

转换　Select the entertainer stage name and the count of members from the entertainers table joined with the entertainer members table on entertainer ID in the entertainers table matches entertainer ID in the entertainer members table then inner joined with the entertainer styles table on style ID in the entertainers table matches style ID in the entertainer styles table, then inner joined with the musical styles table on style ID in the entertainer styles table matches style ID in the musical styles table where the style name is ‘Jazz,’ grouped by entertainer stage name, and having the count of the members greater than 3（依次基于演唱组合 ID 相同将演唱组合表内连接到演唱组合成员表、基于风格 ID 相同内连接到演唱组合风格表、基于风格 ID 相同内连接到音乐风格表；选择风格名为 Jazz 的行并按演唱组合艺名分组；选择成员超过 3 人的演唱组合的艺名和成员数）

整理　Select ~~the~~ ent~~ertainer~~ stage name ~~and the~~ count(*) ~~of members~~ as CountOfMembers from ~~the~~ entertainers ~~table~~ inner join~~ed with the~~ entertainer members ~~table~~ on entertainers. entertainer ID ~~in the entertainers table matches~~ = entertainer_members.entertainer ID ~~in the entertainer members table then~~ inner join~~ed with the~~ entertainer styles ~~table~~ on entertainers.style ID ~~in the entertainers table matches~~ = entertainer_styles.style ID ~~in the entertainer styles table, then~~ inner join~~ed with the~~ musical styles ~~table~~ on entertainer_styles.style ID ~~in the entertainer styles table matches~~ = musical_styles.style ID ~~in the musical styles table~~ where ~~the~~ style name ~~is~~ = ‘Jazz’~~,~~ group~~ed~~ by entertainer stage name~~, and~~ having ~~the~~ count(*) ~~of the members greater than~~ > 3

SQL

```
SELECT Entertainers.EntertainerID,
   Entertainers.EntStageName,
   Count(Entertainer_Members.EntertainerID)
   AS CountOfMembers
FROM ((Entertainers INNER JOIN Entertainer_Members
ON Entertainers.EntertainerID =
   Entertainer_Members.EntertainerID)
INNER JOIN Entertainer_Styles
ON Entertainers.EntertainerID =
   Entertainer_Styles.EntertainerID)
INNER JOIN Musical_Styles
ON Musical_Styles.StyleID =
   Entertainer_Styles.StyleID
WHERE Musical_Styles.StyleName = 'Jazz'
GROUP BY Entertainers.EntertainerID,
   Entertainers.EntStageName
HAVING
Count(Entertainer_Members.EntertainerID) > 3
```

虽然我在最终输出中包含了成员数，但在 HAVING 子句中无须使用 COUNT(*)。在 HAVING 子句中，使用的计算值或列引用只要能够从行组派生出来，就没有任何问题。我将这个查询保存到了示例数据库中，并命名为 CH14_Jazz_Entertainers_More_Than_3。

14.2　在哪里筛选更好

现在，你知道两种筛选最终结果集的方式：WHERE 和 HAVING；你还知道，对于可在 HAVING 子句的查找条件中使用的谓词，存在一些限制。在有些情况下，可将谓词放在 WHERE 子句中，也可将其放在 HAVING 子句中。我们来看看将筛选器放在 WHERE 子句而不是 HAVING 子句中的原因。

14.2.1　在 WHERE 还是 HAVING 子句中筛选

第 6 章介绍过，可使用 5 大类谓词来筛选 FROM 子句返回的行，它们是比较谓词（=、<>、<、>、>=、<=）、范围谓词（BETWEEN）、集合成员资格谓词（IN）、模式匹配谓词（LIKE）和空值判断谓词（IS NULL）。第 11 章拓展

了这方面的知识，演示了如何将子查询用作比较谓词和集合成员资格谓词的参数；还介绍了另外两类需要将子查询作为参数的谓词：限定谓词（ANY、SOME、ALL）和存在谓词（EXISTS）。

别忘了，WHERE 子句中的查找条件在数据库系统进行分组前筛选行。一般而言，最终只需将部分行分组时，最好先在 WHERE 子句中将不需要的行排除在外。例如，假设你要解决如下问题：

“显示位于美国西海岸且订单总价超过 100 万美元的州。”

图 14-3 列出了为解决这个问题需要用到的表。

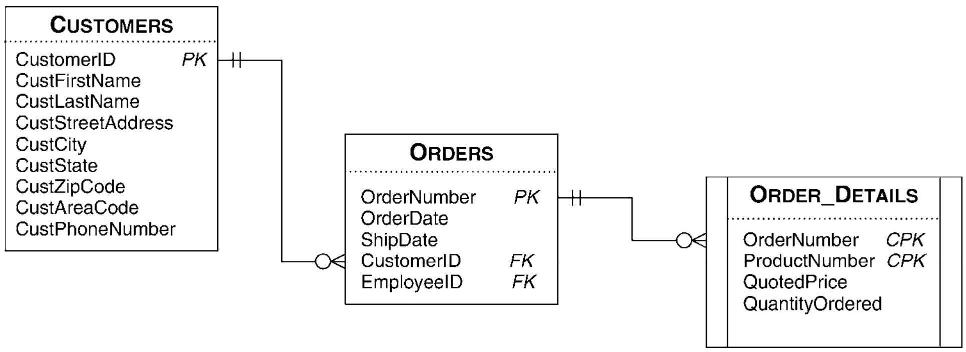


图 14-3　为计算各州的订单总价需要用到的表

可像下面这样将检查顾客所在州的谓词放在 HAVING 子句中来解决这个问题，这是完全合法的。

SQL

```
SELECT Customers.CustState,
   SUM(Order_Details.QuantityOrdered *
   Order_Details.QuotedPrice) AS SumOfOrders
FROM (Customers
   INNER JOIN Orders
   ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber =
   Order_Details.OrderNumber
GROUP BY Customers.CustState
HAVING SUM(Order_Details.QuantityOrdered *
   Order_Details.QuotedPrice) > 1000000
AND CustState IN ('WA', 'OR', 'CA')
```

由于你要根据包含州的列分组，因此可在 HAVING 子句中使用一个检查该列的谓词，但这可能导致数据库系统做不必要的工作。事实上，得克萨斯州的顾客订单总价也超过了 100 万美元。如果像这里这样将基于顾客所在州的筛选器放在 HAVING 子句中，数据库系统将计算得克萨斯州顾客的订单总价、评估第一个谓词，并最终发现得克萨斯州并非是你想要的而将其丢弃。在我提供的示例数据库中，只有位于 CA、TX、OR 和 WA 等州的顾客，但可以想见，如果数据库中包含位于全部 50 个州的顾客，这种性能问题将有多严重：数据库系统必须为所有的州执行计算，然后丢弃除 3 个州外的其他所有州的计算结果。

如果要根据顾客所在的州分组并计算结果，但只考虑华盛顿州、俄勒冈州和加州的顾客，那么这样做更合理：在使用 GROUP BY 按州进行分组前，先使用 WHERE 筛选出与这 3 个州相关的行。如果不这样做，FROM 子句将返回与所有州相关的行，导致数据库系统必须做额外的工作——将你根本不需要的行分组。下面是解决这个问题的更好的方式：

转换	Select customer state and the sum of quantity ordered times quoted price as SumOfOrders from the customers table joined with the orders table on customer ID in the customers table matches customer ID in the orders table, and then joined with the order details table on order number in the orders table matches order number in the order details table where customer state is in the list ‘WA’, ‘OR’, ‘CA’, grouped by customer state, and having the sum of the orders greater than $1 million（依次基于顾客 ID 相同将顾客表内连接到订单表、基于订单号相同内连接到订单详情表；找出顾客所在州为 WA、OR 或 CA 的行，按顾客所在的州分组；找出订单总价超过 100 万美元的组，并选择顾客所在的州以及订购数量与报价乘积总和）

整理

Select cust~~omer~~ state, ~~and the~~ sum ~~of~~ (quantity ordered ~~times~~ * quoted price) as SumOfOrders from ~~the~~ customers ~~table~~ inner join~~ed with the~~ orders ~~table~~ on customers. customer ID ~~in the customers table matches~~ = orders. customer ID ~~in the orders table, and then~~ join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table matches~~ = order_details.order number ~~in the order details table~~ where cust~~omer~~ state ~~is~~ in ~~the list~~ ('WA', 'OR', 'CA')~~,~~ group~~ed~~ by customer state~~, and~~ having ~~the~~ sum ~~of the orders~~ (quantity ordered * quoted price) ~~greater than~~ > ~~$1 million~~ 1000000

SQL

```
SELECT Customers.CustState,
   SUM(Order_Details.QuantityOrdered *
   Order_Details.QuotedPrice) AS SumOfOrders
FROM (Customers
   INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber =
   Order_Details.OrderNumber
WHERE Customers.CustState IN ('WA', 'OR', 'CA')
GROUP BY Customers.CustState
HAVING SUM(Order_Details.QuantityOrdered *
   Order_Details.QuotedPrice) > 1000000
```

请注意，你必须在 HAVING 子句再次输入同样的表达式，而不能使用给它指定的别名。我将这个查询保存到了相应的示例数据库中，并将其命名为 CH14_West_Coast_Big_Order_States。

14.2.2　避开 HAVING COUNT 陷阱

你常常想知道哪些类别的成员数量小于特定的值，例如，你可能想知道哪些演唱组合的成员数少于 3 个、哪些菜品使用的奶制品食材少于 3 种或讲授哪些科目的全职教员少于 4 人。这里的陷阱是必须将成员数为零的类别也包括在内。

我们来看一个问题，它演示了你可能落入的陷阱：

“指出讲授哪些课程类别的全职教授少于 3 人。”

图 14-4 列出了为解决这个问题需要用到的表。

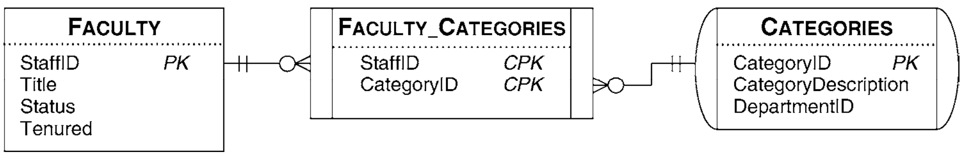


图 14-4　为找出讲授哪些课程类别的教授少于 3 人需要用到的表

转换

Select category description and the count of staff ID as ProfCount from the categories table joined with the faculty categories table on category ID in the categories table matches category ID in the faculty categories table, and then joined with the faculty table on staff ID in the faculty table matches staff ID in the faculty categories table where title is 'Professor,' grouped by category description, and having the count of staff ID less than 3［依次基于类别 ID 相同将类别表内连接到教员类别表、基于教员 ID 相同内连接到全体教员表；选择职称为 Professor 的行，按类别描述分组；从教员 ID 数小于 3 的分组中选择类别描述以及教员 ID 数（作为 ProfCount）］

整理

Select category description ~~and the~~ count ~~of~~ (staff ID) as ProfCount from ~~the~~ categories ~~table~~ inner join~~ed with the~~ faculty categories ~~table~~ on categories.category ID ~~in the categories table matches~~ = faculty_categories. category ID ~~in the faculty categories table, and then~~ inner join~~ed with the~~ faculty ~~table~~ on faculty.staff ID ~~in the faculty table matches~~ = faculty_categories.staff ID ~~in the faculty categories table~~ where title ~~is~~ = 'Professor~~,~~' group~~ed~~ by category description~~, and~~ having ~~the~~ count ~~of~~ (staff ID) ~~less than~~ < 3

SQL

```
SELECT Categories.CategoryDescription,
   COUNT(Faculty_Categories.StaffID) AS
   ProfCount
FROM (Categories
INNER JOIN Faculty_Categories
ON Categories.CategoryID =
   Faculty_Categories.CategoryID)
INNER JOIN Faculty
ON Faculty.StaffID = Faculty_Categories.StaffID
WHERE Faculty.Title = 'Professor'
GROUP BY Categories.CategoryDescription
HAVING COUNT(Faculty_Categories.StaffID) < 3
```

看起来很不错，不是吗？下面是这个查询返回的结果集。

CH14_Subjects_Fewer_3_Professors_WRONG

CategoryDescription	ProfCount
Accounting	1
Business	2
Computer Information Systems	1
Economics	1
Geography	1
History	1
Journalism	1
Math	1
Political Science	1

注意，该结果集中不包含没有教授的科目类别。怎么会这样呢？因为函数 COUNT 只计算根据全职教授进行筛选后 Faculty_Categories 表中余下的行数。换而言之，WHERE 子句将不包含任何行的类别排除在外了。

为了证实我的猜测——有些科目类别没有全职教授，我们编写一个查询来查找没有全职教授的科目类别。别忘了，对于空集合，聚合函数 COUNT 返回零，因此如果让查询计算与科目类别相关的行数，可能会得到空集合。为此，我让查询每次检查一个科目类别，并计算与科目类别（而不是教员）相关的行数。请看下面的 SELECT 语句：

```
SQL          SELECT COUNT(Faculty.StaffID)
             AS BiologyProfessors
             FROM (Faculty
             INNER JOIN Faculty_Categories
             ON Faculty.StaffID =
                Faculty_Categories.StaffID)
             INNER JOIN Categories
             ON Categories.CategoryID =
                Faculty_Categories.CategoryID
             WHERE Categories.CategoryDescription =
                'Biology'
             AND Faculty.Title = 'Professor'
```

BiologyProfessors
0

我将这个查询保存到了相关的示例数据库中，并将其命名为 CH14_Count_Of_Biology_Professors。如你所见，在示例数据库 School Scheduling 中，确实没有讲授生物学的全职教授。我让这个查询只考虑一个科目类别；由于没有职称为 Professor 且科目类别描述为 Biology 的行，因此得到了一个空集，而函数 COUNT 返回零。

确定这一点后，便可将这个查询作为子查询嵌入到 WHERE 子句中，它提取与外部查询提供的类别 ID 匹配的行。这迫使查询每次考虑一个类别，因为这个子查询每次从外部查询指定的 Categories 表中取出一行。相应的 SQL 如下：

```
SQL      SELECT Categories.CategoryDescription,
           (SELECT COUNT(Faculty.StaffID)
            FROM (Faculty
            INNER JOIN Faculty_Categories
            ON Faculty.StaffID =
               Faculty_Categories.StaffID)
            INNER JOIN Categories AS C2
            ON C2.CategoryID =
               Faculty_Categories.CategoryID
            WHERE C2.CategoryID = Categories.CategoryID
              AND Faculty.Title = 'Professor')
            AS ProfCount
         FROM Categories
```

```
WHERE
  (SELECT COUNT(Faculty.StaffID)
   FROM (Faculty
   INNER JOIN Faculty_Categories
   ON Faculty.StaffID =
      Faculty_Categories.StaffID)
   INNER JOIN Categories AS C3
   ON C3.CategoryID =
      Faculty_Categories.CategoryID
   WHERE C3.CategoryID = Categories.CategoryID
   AND Faculty.Title = 'Professor') < 3
```

我将这个查询保存到了相关的示例数据库中，并将其命名为 CH14_Subjects_Fewer_3_Professors_RIGHT。请注意，我在 SELECT 子句中复制了这个子查询，以便显示每个科目类别的全职教授数量。这条 SQL 语句是正确的，因为对于没有全职教授的科目类别，WHERE 子句返回零。正确的结果如下：

CH14_Subjects_Fewer_3_Professors_RIGHT

CategoryDescription	ProfCount
Accounting	1
Biology	0
Business	2
Chemistry	0
Computer Information Systems	1
Computer Science	0
Economics	1
Geography	1
History	1
Journalism	1
Math	1
Physics	0
Political Science	1
Psychology	0
French	0
German	0

如你所见，很多科目类别实际上都没有全职教授。这个最终的解决方案根本没有使用 HAVING，这里介绍它旨在让你知道，对于这种类型的问题，使用 HAVING 的解决方案并非总是清晰的。别忘了，对于很多“……少于……”问题，依然可使用 HAVING 来解决。例如，如果要列出上个月的花费少于 500 美元的所有顾客，但不关心那些什么都没买的顾客，使用 HAVING 可很好地解决问题（执行速度也很可能更快）。然而，如果你要同时列出什么都没买的顾客，就必须采用刚才介绍的这种不使用 HAVING 的方法。

话虽这样说，但实际上使用 GROUP BY 和 HAVING 是能够解决这个问题的。第 13 章演示了如何在查询中获取计数为零的分组，这个查询将演唱组合表外连接到了演出合约表。为了解决刚才说的科目类别和教授问题，诀窍是在 FROM 子句中使用一个筛选出教授的子查询，这样执行连接操作时使用的结果集将是筛选过的。我将这个解决方案留给你去完成，详情请参阅 14.6 节。不用担心，这个解决方案可在相应的示例数据库中找到。

14.3 HAVING 的用途

至此，你对如下主题应该有了深入的认识：如何使用聚合函数和 GROUP BY 子句获取分组小计；如何使用 HAVING 筛选分组数据。要想展示 HAVING 的用途有多广泛，最佳的方式是列举一些使用这个子句可解决的问题（有关该子句的使用示例，请参阅 14.4 节）。

“列出满足如下条件的供应商及其平均可在多少天内将商品送达：送达商品所需的平均天数超过全部供应商送达商品所需的平均天数。”

“显示满足如下条件的商品的名称和总销量：总销量高于所属类别全部商品的平均销量。”

“列出满足如下条件的订单的顾客姓名、下单日期和订购的商品总价：订购的商品总价超过 1000 美元。”

“有多少订单只包含一种商品？”

“哪些经纪人签订的 2017 年 12 月的演出合约的总价格超过了 3000 美元？”

“哪些演唱组合有多个演出合约的时间是重叠的。”

“列出满足如下条件的经纪人的姓名、签订的演出合约价格总和以及总佣金：总佣金超过 1000 美元。”

“有原始得分比球队的其他队员都高的队长吗？”

“显示平均原始得分高于 155 的投球手的姓名和平均原始得分。”

“列出最高原始得分至少比其平均原始得分高 20 的投球手。”

“按类别和学生列出类别名以及满足如下条件的学生的姓名和相应类别课程的平均成绩：选修了相应类别中已结束的课程，且平均成绩不低于 90。”

“按类别显示至少包含 3 门课程的类别的名称及其包含的课程数量。”

“列出满足如下条件的教员及其讲授的课程数量：讲授的课程至少 1 门，但少于 3 门。”

“列出同时使用了牛肉和大蒜的菜品。”

“计算各类菜品使用的总盐量，并显示需要的总盐量超过 3 茶匙的菜品类型。”

“哪些菜品类型包含的菜品不少于两个。”

14.4 语句举例

知道使用 GROUP BY 子句编写查询的机制以及可使用它来解决的一些问题类型后，下面来看一组示例，它们都要求对信息进行分组并根据聚合值进行筛选，且都来自示例数据库。

❖ **说明** 本书前言提醒过，在你使用的数据库系统中，行的排列顺序不一定与本书示例中显示的相同——除非使用 ORDER BY 子句。即便指定了排列方式，在数据库系统返回的结果中，对于 ORDER BY 子句中未包含的列，根据它们的排列顺序也可能不同，这是因为优化方法因数据库系统而异。

如果你在 Microsoft SQL Server 中运行示例，则从视图中选择行时，将不会遵循其中的 ORDER BY 子句指定的排列顺序。要遵循 ORDER BY 子句指定的排列顺序，必须打开视图的设计，并在显示界面中执行它。

另外，当你使用 GROUP BY 子句时，数据库系统返回的结果集看起来像是根据该子句中指定的列进行了排序。这是因为有些优化器首先会在内部对数据排序，以提高 GROUP BY 的处理速度。别忘了，如果要按特定的顺序排列，必须使用 ORDER BY 子句。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库中，查询名以 CH14 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

❖ **说明** 别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录 B。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。这些示例都假定你深入研究并掌握了本书前面介绍的概念，尤其是连接和子查询。

- **Sales Orders 数据库**

“列出满足如下条件的订单的顾客姓名、下单日期和订购的商品总价：订购的商品总价超过 1000 美元。”

转换/整理

Select cust~~omer~~ first name ~~and~~ || ' ' || cust~~omer~~ last name as CustFullName, order date, ~~and the~~ sum ~~of~~ (quoted price ~~times~~ * quantity ordered) as TotalCost from ~~the~~ customers ~~table~~ inner join~~ed with the~~ orders ~~table~~ on customers. customer ID ~~in the customers table matches~~ = orders.customer ID ~~in the orders table, and then~~ inner join~~ed with the~~ order details table on orders.order number ~~in the orders table matches~~ = order_details.order number ~~in the order details table,~~ group~~ed~~ by customer first name, cust~~omer~~ last name, ~~and~~ order date~~,~~ having ~~the~~ sum ~~of~~ (quoted price ~~times~~ * quantity ordered) ~~greater than~~ > 1000［依次基于顾客 ID 相同将顾客表内连接到订单表、基于订单号相同内连接到订单详情表；按顾客的名、姓以及下单日期分组，从报价与订购数量的乘积总和大于 1000 的分组中，选择顾客的名和姓（将其拼合起来作为 CustFullName）、下单日期以及报价与订购数量的乘积总和（作为 TotalCost）］

SQL

```
SELECT Customers.CustFirstName || ' ' ||
   Customers.CustLastName AS CustFullName,
   Orders.OrderDate,
   SUM(Order_Details.QuotedPrice *
    Order_Details.QuantityOrdered)
    AS TotalCost
FROM (Customers
INNER JOIN Orders
ON Customers.CustomerID = Orders.CustomerID)
INNER JOIN Order_Details
ON Orders.OrderNumber =
   Order_Details.OrderNumber
GROUP BY Customers.CustFirstName,
   Customers.CustLastName, Orders.OrderDate
HAVING SUM(Order_Details.QuotedPrice *
   Order_Details.QuantityOrdered) > 1000
```

CH14_Order_Totals_By_Customer_And_Date_GT1000（649 行）

CustFullName	OrderDate	TotalCost
Alaina Hallmark	2017-09-03	$4,699.98
Alaina Hallmark	2017-09-15	$4,433.95
Alaina Hallmark	2017-09-22	$3,951.90
Alaina Hallmark	2017-09-23	$10,388.68
Alaina Hallmark	2017-10-01	$3,088.00
Alaina Hallmark	2017-10-13	$6,775.06
Alaina Hallmark	2017-10-23	$15,781.10
Alaina Hallmark	2017-10-31	$15,969.50
	<< 其他行 >>	

- **Entertainment Agency 数据库**

“哪些经纪人签订的 2017 年 12 月的演出合约的总价格超过了 3000 美元？”

转换/整理

Select ~~the~~ agent first name, agent last name, ~~and the~~ sum ~~of~~ (contract price) as TotalBooked from ~~the~~ agents ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on agents.agent ID ~~in the agents table matches~~ = engagements.agent ID ~~in the engagements table~~ where ~~the~~ engagement start date ~~is~~ between ~~December 1, 2017,~~ ‘2017-12-01’ and ~~December 31, 2017,~~ ‘2017-12-31’, group~~ed~~ by agent first name, ~~and~~ agent last name~~, and~~ having ~~the~~ sum ~~of~~ (contract price) ~~greater than~~ > 3000［基于经纪人 ID 相同将经纪人表内连接到演出合约表；选择演出合约开始日期在 2071 年 12 月 1 日和 12 月 31 日之间的行，并按经纪人的名和姓分组；从合约总价超过 3000 的分组中选择经纪人的名和姓以及合约总价（作为 TotalBooked）］

SQL

```
SELECT Agents.AgtFirstName, Agents.AgtLastName,
   SUM(Engagements.ContractPrice)
   AS TotalBooked
FROM Agents
INNER JOIN Engagements
ON Agents.AgentID = Engagements.AgentID
WHERE Engagements.StartDate
BETWEEN '2017-12-01' AND '2017-12-31'
GROUP BY Agents.AgtFirstName, Agents.AgtLastName
HAVING SUM(Engagements.ContractPrice) > 3000
```

CH14_Agents_Book_Over_3000_12_2017（2 行）

AgtFirstName	AgtLastName	TotalBooked
Marianne	Weir	$6,000.00
William	Thompson	$3,340.00

❖ **警告**　如果你使用的数据库系统使用的是同时存储日期和时间的数据类型，使用 BETWEEN 的查找条件可能不会像预期的那样起作用，因为用户可能在你期望只包含日期的列中输入了日期和时间（在 Microsoft Office Access 中，我不得不使用数据类型 Date/Time，因此在相关的数据库示例中，我只输入了日期）。当日期和时间列同时包含日期和时间时，该列的值将比只包含其中的日期部分时大。例如，2017-12-31 12:00:00 大于 2017-12-31，因此使用 BETWEEN 的查找条件将不会取回相应的行。如果你怀疑列中同时包含日期和时间，应像下面这样编写前述查找条件：

```
StartDate >= '2017-12-01' AND StartDate < '2018-01-01'
```

其中的第二个条件确保你将取回所有与 2017 年 12 月 31 日相关的行，即便它们包含时间部分。

- **School Scheduling 数据库**

“按类别和学生列出类别名以及满足如下条件的学生的姓名和相应类别课程的平均成绩：选修了相应类别中已结束的课程，且平均成绩不低于 90。”

转换/整理

Select category description, stud~~ent~~ first name, stud~~ent~~ last name, ~~and the average~~ avg(~~of~~ grade) as AvgOfGrade from ~~the~~ categories ~~table~~ inner join~~ed with the~~ subjects ~~table~~ on categories.category ID ~~in the categories table matches~~ = subjects.category ID ~~in the subjects table, then~~ inner join~~ed with the~~ classes ~~table~~ on subjects.subject ID ~~in the subjects table matches~~ = classes. subject ID ~~in the classes table, then~~ inner join~~ed with the~~ student schedules ~~table~~ on classes.class ID ~~in the classes table matches~~ = student_schedules.class ID ~~in the student schedules table, then~~ inner join~~ed with the~~ student class status ~~table~~ on student_class_status.class status ~~in the student class status table matches~~ = student_schedules.class status ~~in the student schedules table, and finally~~ inner join~~ed with the~~ students ~~table~~ on students.student ID ~~in the students table matches~~ = student_ schedules.student ID ~~in the student schedules table~~ where class status description is = ‘Completed,’ group~~ed~~ by category description, stud~~ent~~ first name, ~~and~~ stud~~ent~~ last name, ~~and~~ having ~~the average~~ avg(~~of~~ grade) ~~greater than~~ > 90［依次基于类别 ID 相同将类别表内连接到科目表、基于科目 ID 相同内连接到课程表、基于课程 ID 相同内连接到学生选课表、基于课程状态相同内连接到学生课程状态表、基于学生 ID 相同内连接到学生表；选择课程状态描述为 Completed 的行，并按类别描述、学生的名和姓分组；从平均成绩超过 90 的分组中选择类别描述、学生的名和姓以及平均成绩（作为 AvgOfGrade）］

SQL

```
SELECT Categories.CategoryDescription,
   Students.StudFirstName,
   Students.StudLastName,
   AVG(Student_Schedules.Grade) AS AvgOfGrade
FROM ((((Categories
INNER JOIN Subjects
ON Categories.CategoryID = Subjects.CategoryID)
INNER JOIN Classes
ON Subjects.SubjectID = Classes.SubjectID)
INNER JOIN Student_Schedules
ON Classes.ClassID = Student_Schedules.ClassID)
INNER JOIN Student_Class_Status
ON Student_Class_Status.ClassStatus =
   Student_Schedules.ClassStatus)
INNER JOIN Students
ON Students.StudentID =
   Student_Schedules.StudentID
WHERE Student_Class_Status.ClassStatusDescription =
   'Completed'
GROUP BY Categories.CategoryDescription,
   Students.StudFirstName,
   Students.StudLastName
HAVING AVG(Student_Schedules.Grade) > 90
```

CH14_A_Students（17 行）

CategoryDescription	StudFirstName	StudLastName	AvgOfGrade
Accounting	Elizabeth	Hallmark	91.12
Art	Kerry	Patterson	99.83
Biology	Brannon	Jones	94.54
Biology	Karen	Smith	93.05
Chemistry	Richard	Lum	98.31
Computer Information Systems	Janice	Galvin	90.56
Computer Information Systems	John	Kennedy	92.36
Computer Information Systems	Steve	Pundt	98.01
English	Brannon	Jones	91.66
English	Janice	Galvin	91.44
	<< 其他行 >>		

“列出满足如下条件的教员及其讲授的课程数量：讲授的课程至少 1 门，但少于 3 门。”

❖ **说明**　为避免 HAVING COUNT 处理零时出现的问题，我故意指出要获取至少讲授 1 门课程的教员。

转换/整理

Select sta~~ff~~ first name, sta~~ff~~ last name, ~~and the~~ count ~~of classes~~ (*) as ClassCount from ~~the~~ staff ~~table~~ inner join~~ed with the~~ faculty classes ~~table~~ on staff.staff ID ~~in the staff table matches~~ = faculty_classes.staff ID ~~in the faculty classes table,~~ group~~ed~~ by sta~~ff~~ first name, ~~and~~ sta~~ff~~ last name~~, and~~ having ~~the~~ count ~~of classes~~ (*) ~~less than~~ < 3［基于教员 ID 相同将教员表内连接到教员课程表；按教员的名和姓分组，并从课程数少于 3 的分组中选择教员的名和姓以及课程数（作为 ClassCount）］

SQL

```
SELECT Staff.StfFirstName, Staff.StfLastName,
   COUNT(*) AS ClassCount
FROM Staff
INNER JOIN Faculty_Classes
ON Staff.StaffID = Faculty_Classes.StaffID
GROUP BY Staff.StfFirstName, Staff.StfLastName
HAVING COUNT(*) < 3
```

CH14_Staff_Class_Count_1_To_3（2 行）

StfFirstName	StfLastName	ClassCount
Luke	Patterson	2
Michael	Hernandez	2

- **Bowling League 数据库**

“列出最高原始得分至少比其平均原始得分高 20 的投球手。”

转换/整理

Select bowler first name, bowler last name, ~~the average~~ avg(raw score) as CurrentAverage, ~~and the~~ maxi~~mum~~(raw score) as HighGame from ~~the~~ bowlers ~~table~~ inner join~~ed with the~~ bowler scores ~~table~~ on bowlers.bowler ID ~~in the bowlers table matches~~ = bowler_scores.bowler ID ~~in the bowler scores table,~~ group~~ed~~ by bowler first name, ~~and~~ bowler last name~~, and~~ having ~~the~~ maxi~~mum~~ (raw score) ~~greater than~~ > ~~the average~~ avg(raw score) ~~plus~~ + 20［基于投球手 ID 相同将投球手表内连接到投球手得分表，并按投球手的名和姓分组；从最高原始得分比平均原始得分加 20 还高的分组中，选择投球手的名和姓、平均原始得分（作为 CurrentAverage）和最高原始得分（作为 HighGame）］

SQL

```
SELECT Bowlers.BowlerFirstName,
       Bowlers.BowlerLastName,
   AVG(Bowler_Scores.RawScore) AS CurrentAverage,
   MAX(Bowler_Scores.RawScore) AS HighGame
FROM Bowlers INNER JOIN Bowler_Scores
ON Bowlers.BowlerID = Bowler_Scores.BowlerID
GROUP BY Bowlers.BowlerFirstName,
         Bowlers.BowlerLastName
HAVING MAX(Bowler_Scores.RawScore) >
   (AVG(Bowler_Scores.RawScore) + 20)
```

CH14_Bowlers_Big_High_Score（15 行）

BowlerFirstName	BowlerLastName	CurrentAverage	HighGame
Alaina	Hallmark	158	180
Angel	Kennedy	163	194
Ben	Clothier	163	192
Caleb	Viescas	164	193
David	Fournier	157	178
David	Viescas	168	195
Gary	Hallmark	157	179
John	Kennedy	166	191
John	Viescas	168	193
<< 其他行 >>			

- **Recipes 数据库**

“列出同时使用了牛肉和大蒜的菜品。”

转换/整理

Select recipe title from ~~the~~ recipes ~~table~~ where ~~the~~ recipe ID ~~is~~ in ~~the~~ (select~~ion of~~ recipe ID from ~~the~~ ingredients ~~table~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on recipe_ingredients.ingredient ID ~~in the recipe ingredients table matches~~ = ingredients.ingredient ID ~~in the ingredients table~~ where ~~the~~ ingredient name ~~is~~ = ‘Beef’ or ~~the~~ ingredient name ~~is~~ = ‘Garlic,’ group~~ed~~ by recipe ID ~~and~~ having ~~the~~ count ~~of the values in~~ (recipe ID) ~~equal to~~ = 2)（基于食材 ID 相同将食材表内连接到菜品食材表，选择食材名为 Beef 或 Garlic 的行；按菜品 ID 分组，并从菜品 ID 数为两个的分组中选择菜品 ID，生成一个菜品 ID 列表；在菜品表中，从菜品 ID 位于上述列表中的行中选择菜品名）

SQL

```
SELECT Recipes.RecipeTitle
FROM Recipes
WHERE Recipes.RecipeID
IN (SELECT Recipe_Ingredients.RecipeID
   FROM Ingredients
   INNER JOIN Recipe_Ingredients
   ON Ingredients.IngredientID =
      Recipe_Ingredients.IngredientID
   WHERE Ingredients.IngredientName = 'Beef'
  OR Ingredients.IngredientName = 'Garlic'
   GROUP BY Recipe_Ingredients.RecipeID
   HAVING COUNT(Recipe_Ingredients.RecipeID) = 2)
```

CH14_Recipes_Beef_And_Garlic（1 行）

RecipeTitle
Roast Beef

❖ **说明**　这里在子查询中创造性地使用了 GROUP BY 和 HAVING 来找出同时使用了两种食材的菜品。如果菜品没有使用这两种食材，它将不会出现在这个子查询的结果中；如果菜品只使用了其中一种食材，其计数将为 2；仅当菜品同时使用了这两种食材时，计数才为 2。但请小心，如果有菜品同时需要使用蒜沫和整头的大蒜，但不使用牛肉，这种方法将不管用。两种大蒜会导致计数为 2，因此将选择相应的菜品，虽然它没有使用牛肉。

如果你问我为何在需要检查是否同时使用了牛肉和大蒜时使用 OR 运算符，请复习 6.3.1 节。第 8 章介绍了另一种解决这个问题的方式。第 18 章将介绍另一种创造性地解决这个问题的方式。

14.5　小结

本章首先讨论了如何使用 HAVING 根据聚合计算结果来筛选分组。HAVING 是 SELECT 语句中的最后一个子句，本章简要地介绍了它的语法，并举了一个简单的示例。

接下来通过一个示例阐述了在什么情况下应使用 WHERE 子句而不是 HAVING 子句来筛选行。文中指出在可以选择的情况下，最好将筛选器放在 WHERE 子句中。为了让你能够得心应手地使用 HAVING，本章介绍了对可能包含零行结果的分组进行计数时，需要避开的一种常见陷阱，还演示了另一种解决这类问题的方式。

最后，本章阐述了 HAVING 子句为何很有用，并列举了使用该子句可解决的一系列问题，还通过示例演示了如何使用 HAVING 来编写查询。

下一节列举了一些请求，你可独自将它们转换为 SQL 语句。

14.6 练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 列出满足如下条件的供应商及其平均可在多少天内将商品送达：送达商品所需的平均天数超过全部供应商送达商品所需的平均天数。

  提示：需要使用一个子查询来获取全部供应商送达商品所需的平均天数。

  解决方案见 CH14_Vendor_Avg_Delivery_GT_Overall_Avg（5 行）。

  (2) 显示满足如下条件的商品的名称和总销量：总销量高于所属类别全部商品的平均销量。

  提示：为了计算用于比较的值，必须首先使用 SUM 计算类别中所有商品的总销量，再使用 AVG 计算平均销量。

  解决方案见 CH14_Sales_By_Product_GT_Category_Avg（13 行）。

  (3) 有多少订单只包含一种商品？

  提示：需要在 FORM 子句中使用一个内部查询来获取只有一行的订单的订单号，再在外部查询的 SELECT 子句中使用 COUNT 计算行数。

  解决方案见 CH14_Single_Item_Order_Count（1 行）。

- **Entertainment Agency 数据库**

  (1) 告诉我哪些演唱组合有多个演出合约的时间是重叠的。

  提示：使用子查询找出时间重叠的演出合约数超过 2 的演唱组合。第 6 章演示了如何通过比较高效地确定两个范围是否是重叠的。

  解决方案见 CH14_Entertainers_MoreThan_2_Overlap（1 行）。

  (2) 列出满足如下条件的经纪人的姓名、签订的演出合约价格总和以及总佣金：总佣金超过 1000 美元。

  提示：使用第 13 章类似问题的解决方案，并添加一个 HAVING 子句。

  解决方案见 CH14_Agent_Sales_Big_Commissions（4 行）。

- **School Scheduling 数据库**

  (1) 按类别显示至少包含 3 门课程的类别的名称及其包含的课程数量。

  提示：将类别表内连接到科目表，再内连接到课程表；计算行数并添加一个 HAVING 子句来获得最终结果。

  解决方案见 CH14_Category_Class_Count_3_Or_More（14 行）。

  (2) 列出讲授的课程不超过 3 门的教员及其讲授的课程数。

  提示：这里挖了一个 HAVING COUNT 陷阱！请使用子查询。

  解决方案见 CH14_Staff_Teaching_LessThan_3（7 行）。

  (3) 指出讲授哪些课程类别的全职教授少于 3 个。

  14.2.2 节演示了一种使用子查询来解决这个问题的正确方式。现在请你尝试使用连接和 GROUP BY 来正确地解决它。

提示：考虑在 FORM 子句中使用外连接和子查询。

解决方案见 CH14_Subjects_Fewer_3_Professors_Join_RIGHT（16 行）。

(4) 计算每位教员讲授的课程数。

提示：这其实不是一个可使用 HAVING 解决的问题，但你可能试图错误地使用 GROUP BY 和 COUNT(*) 来解决它。

正确的解决方案见 CH14_Staff_Class_Count_Subquery（27 行）和 CH14_Staff_Class_Count_GROUPED_RIGHT（27 行）。错误的解决方案见 CH14_Staff_Class_Count_GROUPED_WRONG（22 行）。

- **Bowling League 数据库**

(1) 有原始得分比球队的其他队员都高的队长吗？

提示：要找出队长的最高原始得分，可根据队长 ID 相同将球队表内连接到投球手表，再内连接到投球手得分表。使用 HAVING 子句将最高得分同使用子查询获得的其他所有球员的最高得分进行比较。

解决方案见 CH14_Captains_Who_Are_Hotshots（0 行，即没有队长保龄球打得比其所有队友都好）。

(2) 显示平均原始得分高于 155 的投球手的姓名和平均原始得分。

提示：需要使用一个简单的 HAVING 子句来将平均值同一个数字字面量进行比较。

解决方案见 CH14_Good_Bowlers（17 行）。

(3) 列出个人平均得分不低于总体平均得分的投球手的姓和名。

提示：12.3 节演示了如何在 WHERE 子句中使用子查询来解决这个问题。现在请你使用 HAVING 来解决。

解决方案见 CH14_Better_Than_Overall_Average_ HAVING（17 行）。

- **Recipes 数据库**

(1) 计算各类菜品使用的总盐量，并显示需要的总盐量超过 3 茶匙的菜品类型。

提示：连接 5 个表以根据盐和茶匙进行筛选，使用 SUM 计算总量，再筛选出使用的总盐量不超过 3 茶匙的菜品类型。

解决方案见 CH14_Recipe_Classes_Lots_Of_Salt（1 行）。

(2) 哪些菜品类型包含的菜品不少于两个？

提示：将菜品类型表内连接到菜品表，计算结果集中的行数，并使用 HAVING 子句保留行数不少于 2 的菜品类型。

解决方案见 CH14_Recipe_Classes_Two_Or_More（4 行）。

第五部分

修改数据集

第 15 章 更新数据集

“世界在变，不停地变，不可避免地变，这是当今社会的决定性因素。”

——艾萨克·阿西莫夫

本章涵盖如下主题：

- 何谓更新
- UPDATE 语句
- UPDATE 的用途
- 语句举例
- 小结
- 练习

通过第二到第四部分的学习你知道，使用 SELECT 语句从表中获取数据既有趣又颇具挑战性，也许有时候挑战性远远超过了趣味性。如果你只是想回答问题，就没有必要阅读这部分了，但现实世界的大多数应用程序不仅要回答复杂的问题，还要让用户能够修改、添加或删除数据。除本书前面一直在介绍的用于检索数据的 SELECT 语句外，SQL 标准还定义了另外三种让你能够修改数据的语句，本章将介绍其中的第一种：UPDATE。

15.1 何谓更新

SELECT 语句让你能够从表中检索数据集。UPDATE 也处理数据集，但它可以用来**修改**一行或多行中一列或多列的值。现在你应该对表达式了如指掌。要修改列的值，只需将一个表达式赋给它。

但你必须小心行事，因为 UPDATE 语句功能强大。大多数时候，你只想更新一行或几行；如果你不小心，可能会修改数千行。为了避免这种问题，我将演示如何先对你编写的语句进行测试。

> ❖ **说明** 所有示例语句和解决方案都可在相应示例数据库的“修改”版本——Sales Orders Modify、Entertainment Agency Modify、School Scheduling Modify 和 Bowling League Modify 中找到。

15.2 UPDATE 语句

实际上，UPDATE 语句比本书前面一直在学习的 SELECT 语句简单得多，它只有 3 个子句：UPDATE、SET 和可选的 WHERE 子句，如图 15-1 所示。

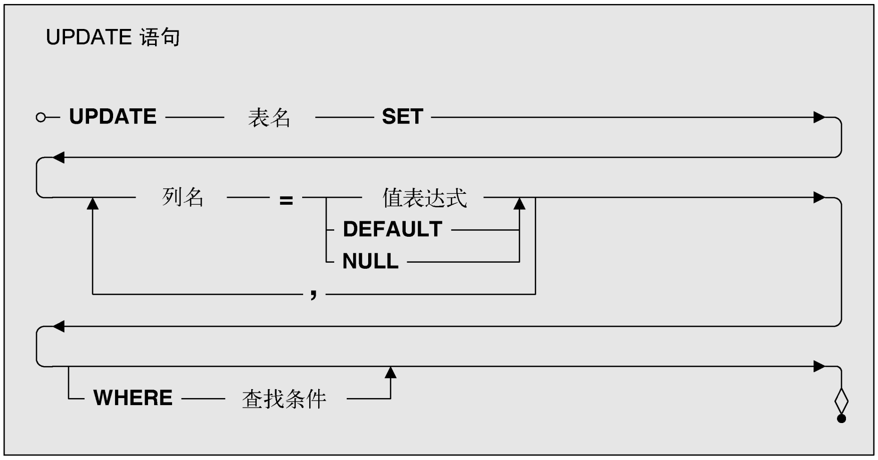


图 15-1　UPDATE 语句的语法图

在关键字 UPDATE 后面，你指定要更新的表；关键字 SET 后面是一个或多个子句，它（们）给表中的列指定新值。必须至少包含一个赋值子句，但可根据需要包含任意数量的赋值子句，以修改每行中多列的值。WHERE 子句是可选的，用于指定要更新目标表中的哪些行。

15.2.1　使用简单的 UPDATE 表达式

我们来看一个示例，它使用简单的表达式给要更新的列赋值。

❖ **说明**　贯穿本章都将采用第 4 章介绍的“请求/转换/整理/SQL”流程。

“将所有商品的零售价提高 10%。”

这有点棘手。你将发现，很难将请求直接转换为类似于 SQL 的英语，因为英语的顺序通常与 UPDATE 要求的顺序不同。请仔细研究这个请求，找出目标表的名称以及要更新的列的名称。我们按表名、列名的顺序来重新组织请求，再进行转换，如下所示。

“修改商品，将零售价提高 10%。”

转换	Update the products table by setting the retail price equal to the retail price plus 10 percent of the price（更新商品表，将零售价设置为原来的 110%）
整理	Update ~~the~~ products ~~table by~~ set~~ting the~~ retail price ~~equal to~~ = ~~the~~ retail price ~~plus~~ + (.1~~0 percent of the~~ * retail price)
SQL	`UPDATE Products` `SET Price = Price + (0.1 * Price)`

请注意，不能使用 SQL 代码 `SET Price + 10 percent`，而必须在等号左边指定要更新的列，再编写一个表达式来计算所需的新值。如果指定新值时，需要用到旧值（当前值），就必须根据需要在等号右边引用列名。在 SQL 标准中，一个非常明确的规则是，更新任何行前，数据库系统必须计算所有的赋值表达式，因此数据库系统将这样解析等号右边对 Price 列的两个引用：先获取 Price 列的值，再修改它。

在所有的编程语言中，这样的赋值语句都很常见。虽然看起来像是将列的值赋给它自己，但实际上是先获取列的值，将其加上 10%，再将结果赋给列，从而将其更新为新值。

1. 更新选定的行

你总是要更新表中所有的行吗？很可能不是这样的。为了限定将被 UPDATE 语句修改的行，需要添加 WHERE 子句。我们再来看一个问题：

“我的服装供应商刚宣布提价 4%，因此需要更新服装类商品的价格，将其提高 4%。”

我们换一种说法：

“修改商品，将服装类商品（3 类）的零售价提高 4%。”

转换	Update the products table by setting the retail price equal to retail price times 1.04 for all products in category 3（更新商品表，将所有 3 类商品的零售价设置为原来的 1.04 倍）
整理	Update ~~the~~ products ~~table by~~ set~~ting the~~ retail price ~~equal to~~ = retail price ~~times~~ * 1.04 ~~for all~~ where ~~products in~~ category ID = 3
SQL	UPDATE Products SET RetailPrice = RetailPrice * 1.04 WHERE CategoryID = 3

❖ **说明**　在这个查询中，我简化了使用的计算公式，将原值乘以 1.04，而不是将原值加上它与 0.04 的乘积。从数学上说，结果没变，但可能会提高执行速度，因为仅执行一次数学运算（价格乘以 1.04）的效率比执行两次（价格加上它与 0.04 的乘积）高。

在第 11 章学习子查询后，这很容易，不是吗？你将在 WHERE 子句中大量地使用子查询，因此后面将对此进行介绍。

2. 安全第一：确保更新的是正确的行

即便是对于简单的更新查询，也强烈建议你核实它将更新正确的行。如何核实呢？前面说过，在大多数情况下，你将添加一个 WHERE 子句，以选择部分行进行更新。为何不先编写一个 SELECT 查询来返回要更新的行呢？接着前面的示例，我们让数据库系统返回一列，以便确保使用正确的值更新正确的行并将表达式赋给要修改的列。

“列出商品表中 3 类商品的名称、零售价及零售价加 4%的结果。”

转换	Select product name, retail price, and retail price times 1.04 from the products table for products in category ID 3（从商品表中选择类别 ID 为 3 的商品的名称、零售价以及零售价与 1.04 的乘积）
整理	Select product name, retail price, ~~and~~ retail price ~~times~~ * 1.04 from ~~the~~ products ~~table for~~ where ~~products in~~ category ID = 3
SQL	SELECT ProductName, RetailPrice, RetailPrice * 1.04 As NewPrice FROM Products WHERE CategoryID = 3

图 15-2 显示了结果。

ProductName	RetailPrice	NewPrice
Ultra-Pro Rain Jacket	$85.00	$88.40
StaDry Cycling Pants	$69.00	$71.76
Kool-Breeze Rocket Top Jersey	$32.00	$33.28
Wonder Wool Cycle Socks	$19.00	$19.76

图 15-2　核实要更新的行

请注意，我包含了商品名，以便能够知道要更新哪些商品。如果这个结果是我想要的，就可将这条 SELECT 语句转换为 UPDATE 语句，方法是删除不需要的元素，并替换 FROM 和 SELECT 子句。图 15-3 说明了如何将这条 SELECT 语句转换为正确的 UPDATE 语句。

只需将不需要的单词删掉，将表名移到 UPDATE 子句中，将目标列和表达式移到 SET 子句中并用等号连接它们，再复制 WHERE 子句，就大功告成了。

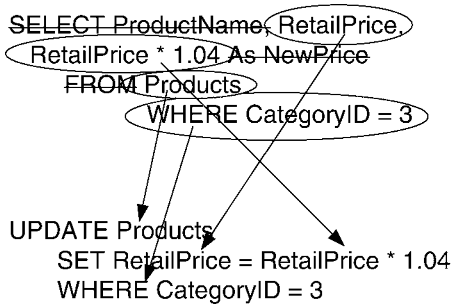


图 15-3 将 SELECT 查询转换为 UPDATE 语句

15.2.2 说点题外话：事务

深入介绍如何修改数据前，需要介绍一个重要的 SQL 功能。SQL 标准定义了**事务**，你可使用它给你对表中数据所做的一系列修改提供保护。SQL 中的事务很像在线交易或到商店购物。你下订单后，就启动了事务。为订购的商品付款是事务的一部分。等你接收了商品后，事务就结束了。但如果商品没有送到，你可能申请退款；如果对商品不满意，你可退货并要求退款。

SQL 标准提供了 3 条语句来模拟这种场景。修改数据前，可使用 START TRANSACTION 来指出你要保护和确保即将要做的修改，这相当于下单。接下来，你修改数据库，这相当于更新付款和收货状态。如果一切顺利，就可使用 COMMIT 来让修改永久化；如果中途出了问题（无法更新付款状态或收货状态），可使用 ROLLBACK 将数据恢复到事务开始前的状态。

这个买卖示例看起来有点傻，但事务是一个非常强大的 SQL 功能，在你需要修改多行或多个表中的行时尤其如此。通过使用事务，可确保所有的修改要么都成功，要么都失败。你不希望在没有收到商品的情况下更新付款状态，也不希望在没有收到款项的情况下将收货状态设置为“已收到”。请注意，上述流程不仅适用于本章介绍的 UPDATE 语句，也适用于接下来的两章将介绍的 INSERT 和 DELETE 语句。

并非所有的数据库系统都实现了事务，而且在不同的数据库系统中，使用事务的语法存在细微的差别。有些数据库系统允许嵌套事务，这让你能够建立多个提交点。有些终端用户数据库系统（如 Microsoft Office Access）在你运行修改数据的查询时，在幕后自动发起事务。如果你使用过 Microsoft Access，就知道它会使用消息指出将修改多少行以及是否有修改会失败，而你可接受修改，也可撤销修改（ROLLBACK）。有关这方面的细节，请参阅你使用的数据库系统的文档。

15.2.3 更新多列

图 15-1 所示的 UPDATE 语句语法图表明，通过使用多条用逗号分隔的赋值语句，可修改多列。别忘了，数据库系统将修改 WHERE 子句返回的所有行。我们来看一种你可能想在 School Scheduling 数据库中执行的更新：

> “修改课程，将所有绘画课（科目 ID 为 13 的课程）的上课地点都改为 1635 号教室，并将上课时间从 Monday-Wednesday-Friday 改为 Tuesday-Thursday-Saturday。”

转换	Update classes and set classroom ID to 1635, MondaySchedule to false, schedule to false, Friday schedule to false, Tuesday schedule to true, Thursday schedule to true, and Saturday schedule to true for all classes that are subject ID 13（更新课程表，对于科目 ID 为 13 的所有课程，将教室 ID 都设置为 1635，将 MondaySchedule、WednesdaySchedule 和 FridaySchedule 都设置为假，并将 TuesdaySchedule、ThursdaySchedule 和 SaturdaySchedule 都设置为真）
整理	Update classes ~~and~~ set classroom ID ~~to~~ = 1635, Monday schedule ~~to~~ = 0 ~~false~~, Wednesday schedule ~~to~~ = 0 ~~false~~, Friday schedule ~~to~~ = 0 ~~false~~, Tuesday schedule ~~to~~ = 1 ~~true~~, Thursday schedule ~~to~~ = 1 ~~true~~, ~~and~~ Saturday schedule ~~to~~ = 1 ~~true for all classes that are~~ where subject ID = 13

```
SQL        UPDATE Classes SET ClassRoomID = 1635,
              MondaySchedule = 0,
              WednesdaySchedule = 0,
              FridaySchedule = 0,
              TuesdaySchedule = 1,
              ThursdaySchedule = 1,
              SaturdaySchedule = 1
           WHERE SubjectID = 13
```

❖ **说明**　别忘了，大多数数据库系统使用 0 来表示假，使用 1 来表示真。有关这方面的详情，请参阅数据库系统的文档。

你可能想要双保险，确保只修改在周一、周三、周五上课的课程，为此，可在 WHERE 子句中再添加一个条件，如下所示。

```
SQL        UPDATE Classes SET ClassRoomID = 1635,
              MondaySchedule = 0,
              WednesdaySchedule = 0, FridaySchedule = 0,
              TuesdaySchedule = 1, ThursdaySchedule = 1,
              SaturdaySchedule = 1
           WHERE SubjectID = 13
              AND MondaySchedule = 1
              AND WednesdaySchedule = 1
              AND FridaySchedule = 1
```

请注意，这里先筛选出你预期列中包含的值，再让 UPDATE 语句修改它。这个修改后的查询查找 SubjectID 为 13 且 MondaySchedule、WednesdaySchedule 和 FridaySchedule 列的值都为真（1）的行。对于满足这些条件的每一行，UPDATE 语句都修改其 ClassRoomID 列和各个 Schedule 列。如果你再次运行这个查询，将发现数据库系统没有更新任何行，这是因为你在第一次运行这个查询时修改了列的值，导致所有行都不满足条件。

15.2.4　使用子查询筛选行

在前几小节的示例中，我更新了 3 类商品以及科目 ID 为 13 的课程。在现实世界中，类似这样的编号值含义不清晰，你更可能会说“服装类商品”或“绘画课”。在 SELECT 查询中，可在 FROM 子句中使用连接添加相关的表，再显示相关表中含义更清晰的值。与往常一样，要执行连接，必须熟悉表间关系。图 15-4 列出了为将类别描述关联到商品需要用到的表。

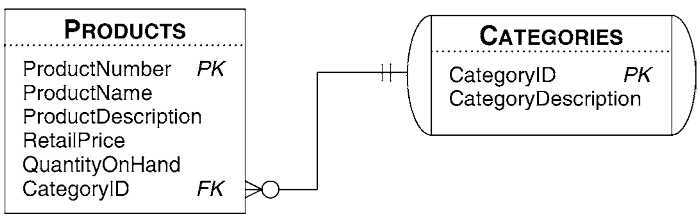


图 15-4　为将类别描述关联到商品需要用到的表

我们再来看前面为检查商品更新而编写的验证查询，但这次再加入 Categories 表：

```
SQL        SELECT ProductName, RetailPrice,
              RetailPrice * 1.04 As NewPrice
           FROM Products INNER JOIN Categories
              ON Products.CategoryID = Categories.CategoryID
           WHERE Categories.CategoryDescription = 'Clothing'
```

相比于使用类别 ID 值 3 进行筛选，使用值 Clothing 进行筛选要合理得多。但请注意，图 15-1 所示的 UPDATE 语句语法图表明，在关键字 UPDATE 后面只能指定一个表，因此无法使用内连接将 Categories 表包含进来，进而根据更

有意义的值进行筛选。如何解决这个问题呢?

第 11 章说过，可在 WHERE 子句中使用筛选器根据从相关表中获取的值进行筛选。我们来再次解决零售价更新问题，但这次使用一个子查询，以便能够根据更有意义的值进行筛选。

“修改商品，将服装类商品的零售价提高 4%。”

转换	Update the products table by setting the retail price equal to retail price times 1.04 for the products whose category ID is equal to the selection of the category ID from the categories table where the category description is clothing（在类别表中，从类别描述为服装的行中选择类别 ID；更新商品表，将类别 ID 与前面选择的类别 ID 相同的商品的零售价设置为原值的 1.04 倍）
整理	Update ~~the~~ products ~~table by~~ set~~ting the~~ retail price ~~equal to~~ = retail price ~~times~~ * 1.04 ~~for the products whose~~ where category ID ~~is equal to~~ = ~~the~~ (selec~~tion of the~~ category ID from ~~the~~ categories ~~table~~ where ~~the~~ category description ~~is~~ = 'Clothing')

SQL	`UPDATE Products`
	`SET RetailPrice = RetailPrice * 1.04`
	`WHERE CategoryID =`
	`    (SELECT CategoryID`
	`   FROM Categories`
	`   WHERE CategoryDescription = 'Clothing')`

这不像根据连接表中的列进行过滤的简单 WHERE 子句那么直观，但毕竟解决了问题。

❖ **警告** 请注意，我将 Products 表的 CategoryID 列同子查询返回的值进行比较，看它们是否相等。第 11 章说过，要在谓词中使用等号与子查询进行比较，子查询必须只返回一个值。如果在 Categories 表中，有多行的类别描述列的值为 Clothing，这个查询将失败，但在这个示例中，我非常确定根据 Clothing 进行筛选将只返回一个 CategoryID 值。如果不确定子查询将只返回一个值，应使用谓词 IN 而不是相等运算符。

下面使用同样的方法来解决课程更新问题。我要使用 Subjects 表中的科目编码或科目名，而不是无意义的数字型科目 ID。图 15-5 列出了需要用到的表。

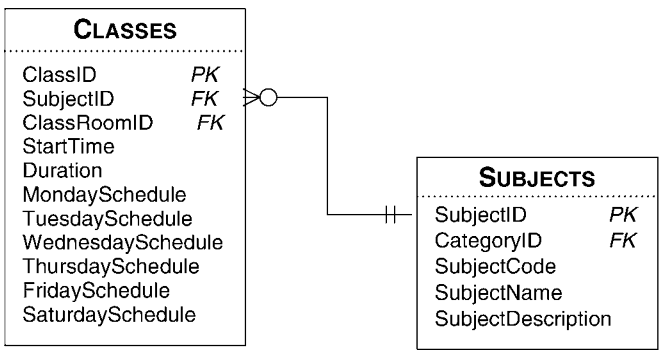


图 15-5 为将科目名关联到课程需要用到的表

我们再次来解决这个更新问题，但使用子查询筛选器。

“修改课程，将所有绘画课的上课地点都改为 1635 号教室，并将上课时间从 Monday-Wednesday-Friday 改为 Tuesday-Thursday-Saturday。”

转换	Update classes and set classroom ID to 1635, Monday schedule to false, Wednesday schedule to false, Friday schedule to false, Tuesday schedule to true, Thursday schedule to true, and Saturday schedule to true for all classes whose subject ID is in the selection of subject IDs from the subjects table where subject name is 'Drawing'（在科目表中，从科目名为 Drawing 的行中选择科目 ID；更新课程表，对于科目 ID 与前面选择的科目 ID 相同的所有课程，都将教室 ID 设置为 1635，将 MondaySchedule、WednesdaySchedule 和 FridaySchedule 都设置为假，并将 TuesdaySchedule、ThursdaySchedule 和 SaturdaySchedule 都设置为真）

整理　Update classes ~~and~~ set classroom ID ~~to~~ = 1635, Monday schedule ~~to~~ = 0 ~~false~~, Wednesday schedule ~~to~~ = 0 ~~false~~, Friday schedule ~~to~~ = 0 ~~false~~, Tuesday schedule ~~to~~ = 1 ~~true~~, Thursday schedule ~~to~~ = 1 ~~true~~, ~~and~~ Saturday schedule ~~to~~ = 1 ~~true for all classes whose~~ where subject ID ~~is~~ in ~~the~~ (selec~~tion of~~ subject ID~~s~~ from ~~the~~ subjects ~~table~~ where subject name ~~is~~ = ‘Drawing’)

SQL

```
UPDATE Classes SET ClassRoomID = 1635,
   MondaySchedule = 0,
   WednesdaySchedule = 0, FridaySchedule = 0,
   TuesdaySchedule = 1, ThursdaySchedule = 1,
   SaturdaySchedule = 1
WHERE SubjectID IN
   (SELECT SubjectID
   FROM Subjects
   WHERE SubjectName = 'Drawing')
```

请注意，虽然我非常确定只有一个科目 ID 对应的科目名为 Drawing，但为安全起见，我还是决定使用谓词 IN。

15.3　有些数据库系统允许在 UPDATE 子句中使用连接

在多种数据库系统（其中最著名的是 Microsoft Access 和 Microsoft SQL Server）中，都可在 UPDATE 查询的 FROM 子句中连接表，但有一个限制，就是必须从一个表的主键连接到另一个表的外键，这样数据库系统就能确定你要更新哪些行。这样，当需要根据相关表中的值筛选行时，你就不用在 WHERE 子句中使用子查询了。

如果你使用的数据库系统允许这样做，就可像下面这样解决修改有关绘画课信息的问题：

SQL

```
UPDATE Classes INNER JOIN Subject
  ON Classes.SubjectID = Subjects.SubjectID
SET ClassRoomID = 1635, MondaySchedule = 0,
   WednesdaySchedule = 0, FridaySchedule = 0,
   TuesdaySchedule = 1, ThursdaySchedule = 1,
   SaturdaySchedule = 1
WHERE Subjects.SubjectName = 'Drawing'
```

如你所见，这样就无须使用子查询来筛选行了。从某种程度上说，这种语法也更容易理解。还可使用这种语法来连接相关的表，以提供要在更新表达式中使用的值，这样就不用在 SET 子句中使用子查询了。请务必查看你使用的数据库系统的文档，看看它是否支持这种功能。你将发现，在 Microsoft Access 版示例数据库中，我使用了这种方法来解决一些示例问题。

顺便说一句，SQL 标准允许目标表为视图，而视图可能是连接表。但这个标准规定，视图更新规则由实现指定，这给数据库系统厂商提供了选择空间：可要求只能使用表名，也可规定你可使用视图或连接表来做什么。有关这方面的细节，请参阅你使用的数据库系统的文档。

可以想见，可根据需要编写非常复杂的子查询，以妥善地筛选目标表。例如，如果你要修改某位教授讲授的课程的开始时间，就需要在子查询的 FROM 子句中连接 Faculty_Classes 表和 Staff 表。图 15-6 列出了需要用到的表。

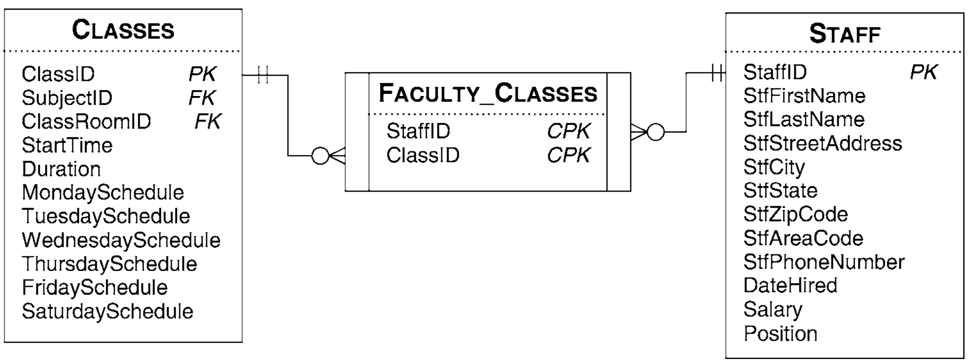


图 15-6　为将教员姓名关联到课程需要用到的表

假设你要将 Kathryn Patterson 讲授的所有课程的开始时间都改为 2:00 PM（你不太可能这样做，因为这可能导致多门课程的开始时间相同，但这是一个很有趣的示例）。我们来看看如何解决这个问题。

"修改课程表，将 Kathryn Patterson 讲授的所有课程的开始时间都设置为 2:00 PM。"

转换	Update the classes table by setting the start time to 2:00 PM for all classes whose class ID is in the selection of class IDs of faculty classes joined with staff on staff ID in the faculty classes table matches staff ID in the staff table where the staff first name is 'Kathryn' and the staff last name is 'Patterson'（基于教员 ID 相同将教员课程表内连接到教员表，从教员名为 Kathryn 且教员姓为 Patterson 的行中选择课程 ID，生成一个课程 ID 列表；更新课程表，将课程 ID 位于前述列表中的所有课程的开始时间都设置为 2:00 PM）
整理	Update ~~the~~ classes ~~table by~~ set~~ting the~~ start time ~~to~~ = ~~2:00 PM~~. '14:00:00' ~~for all classes whose~~ where class ID ~~is~~ in ~~the~~ (selec~~tion of~~ class ID~~s of~~ from faculty classes inner join~~ed with~~ staff on faculty_classes.staff ID ~~in the faculty classes table matches~~ = staff.staff ID ~~in the staff table~~ where ~~the~~ staff first name ~~is~~ = 'Kathryn' and ~~the~~ staff last name ~~is~~ = 'Patterson')

SQL	`UPDATE Classes SET StartTime = '14:00:00'`
	`WHERE ClassID IN`
	`  (SELECT ClassID`
	`  FROM Faculty_Classes INNER JOIN Staff`
	`    ON Faculty_Classes.StaffID = Staff.StaffID`
	`  WHERE StfFirstName = 'Kathryn'`
	`   AND StfLastName = 'Patterson')`

这里的诀窍是，找出在 WHERE 子句中指定条件所需的相关表与目标表的关系，在第 8 章和第 9 章编写多表查询的 FROM 子句时，你这样做过。编写 UPDATE 语句时，只能将目标表放在关键字 UPDATE 后面，因此必须在子查询中使用其他表来返回一个列，进而通过这个列关联到目标表。

使用子查询型更新表达式

如果你以为子查询在 UPDATE 语句中的用途仅限于此，那么你就错了。图 15-1 表明，在 SET 子句中，赋给列的值可以是值表达式。为了帮你复习一下，图 15-7 显示了如何编写值表达式。

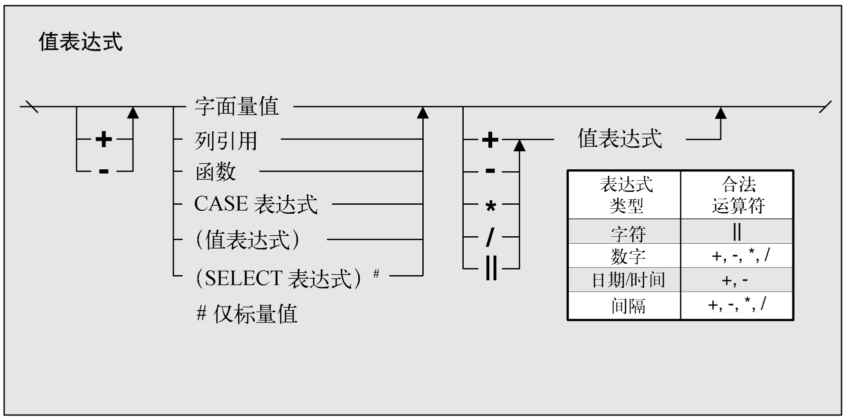


图 15-7　值表达式的语法图

第 2 章建议过，不要在表中包含计算列。与大多数规则一样，这条规则也有例外。请看示例数据库 Sales Orders 中的 Orders 表。如果公司需要超大型订单（对应于数千个订单详情行），你可以考虑在 Orders 表中包含一个订单总价列。通过包含这个计算列，你就可运行查询来查看订单总价，而无须取回数千个订单详情行并计算订单总价。如果你选择这样做，就必须在应用程序中添加这样的代码：每当有相关的订单详情行发生变化时，都重新计算订单总价，确保它是最新的。

❖ **说明**　很多数据库系统都提供了常被称为**触发器**的功能，让你能够在添加、更新或删除数据时运行代码（添加、更新或删除数据的操作“触发”代码）。触发器代码可执行额外的复杂验证，甚至运行额外的更新、插入或删除查询，以修改相关表中的数据。可以想见，你在触发器中编写的代码可能更新相关表中的计算列。

有些数据库系统（其中最著名的是 Microsoft SQL Server）还允许你在表设计中定义计算列。显然，这样的功能导致数据库系统在你处理表中数据时可能做额外的工作，因此使用这样的功能时必须特别小心并尽量少用。有关这方面的细节，请参阅你使用的数据库系统的文档。

在本章前面，在 SET 子句中给列指定的值都是字面量值或由字面量值、运算符和列名组成的值表达式。请注意，你也可指定目标表中另一个列的值，但你想这样做的情况很少。最有趣的是，可使用 SELECT 表达式（子查询）从另一个表返回单个值（如总和），并将其赋给列。你可在这个子查询（的 WHERE 子句）中指定条件，根据要更新的表中的值从另一个表中筛选值。

因此，要使用基于相关表 Order_Details 中列的表达式计算总和，并用计算得到的结果更新 Orders 表中的总价，可运行一个包含子查询的 UPDATE 查询。你使用子查询计算订购数量与报价乘积的总和，再将结果放到相应的计算列中。对于 Orders 表中的每一行，为确保根据 Order_Details 表中的相关行来计算总价，需要添加一个 WHERE 子句。我们来看看如何解决这个问题。

“修改订单表，将订单总价设置为所有相关订单详情行中订购数量与报价乘积的总和。”

转换	Update the orders table by setting the order total to the sum of quantity ordered times quoted price from the order details table where the order number matches the order number in the orders table（更新订单表，将订单总价设置为这样的值：在订单详情表中找出订单号与订单表中当前行的订单号相同的行，并将这些行中订购数量与报价的乘积相加得到的结果）
整理	Update ~~the~~ orders ~~table by~~ set~~ting the~~ order total ~~to~~ = ~~the~~ (select sum ~~of~~ (quantity ordered ~~times~~ * quoted price) from ~~the~~ order details ~~table~~ where ~~the~~ order_details.order number ~~matches the~~ = orders.order number ~~in the orders table~~)
SQL	UPDATE Orders SET OrderTotal = (SELECT SUM(QuantityOrdered * QuotedPrice) FROM Order_Details WHERE Order_Details.OrderNumber = Orders.OrderNumber)

❖ **说明**　我将这个查询保存到了示例数据库 Sales Orders Modify 中，并将其命名为 CH15_Update_Order_Totals_Subquery。

请注意，我没有使用 WHERE 子句来筛选要让数据库系统更新的订单。如果你在应用程序中执行这个查询，可能想筛选订单号，让数据库系统只更新发生了变化的订单。实际上，在有些数据库系统中，可定义计算列并指定数据库系统该如何更新它。前面说过，大多数数据库系统支持触发器。有人在指定表中修改、添加或删除行时，数据库系统会自动运行相应的触发器。在支持这些功能的数据库系统中，你可在表定义中使用这种 UPDATE 查询语法，也可在要在值发生变化后运行的触发器中使用它。有关这方面的细节，请参阅你使用的数据库系统的文档。

15.4　UPDATE 的用途

至此，你应该对如何使用简单的字面量或复杂的子查询表达式来更新一列或多列有了深入的认识，你还知道了如何筛选出将被 UPDATE 语句修改的行。为了展示 UPDATE 语句的用途有多广泛，最佳的方式是列举一些可使用这种语句解决的问题并提供一些使用示例（参阅 15.5 节）。

“对于发货日期比下单日期晚 30 天以上的订单，将报价下调 2%。”

“将所有经纪人的薪水都上调 6%。”

“将原来安排在保龄球馆 Sports World Lanes 举行的所有联赛的比赛地点都改为 Oasis Lanes。”

“基于已结束的课程重新计算所有学生的平均成绩。”

"对于 2017 年 10 月的消费总额超过 50 000 美元的顾客，给其所有订单都打 95 折。"
"更正演出合约的价格，将其设置为演唱组合的每日费用与演出天数的乘积，再加上 15%的佣金。"
"根据每位投球手的邮政编码找出其所在的州和城市并更新这两项。"
"找出邮政编码为 98270 或 98271 的所有学生和教员，将他们的区号都改为 360。"
"确保所有自行车的零售价都比报价最低的供应商提供的批发价至少高 45%。"
"对于 2017 年 10 月的消费总额超过 3000 美元的顾客，给其所有演出合约的价格都打 98 折。"
"将保龄球队 Huckleberrys 改名为 Manta Rays。"
"将全职终身教员的薪水提高 5%。"
"将配饰的零售价设置为报价最高的供应商提供的批发价加 35%。"
"对于总销售额超过 20 000 美元的经纪人，将其佣金比例提高 0.5%。"
"计算并更新所有投球手的总得分、比赛局数、当前平均得分和当前让分数。"

15.5　语句举例

至此，你知道了编写 UPDATE 查询的机制。下面来看一些示例，它们来自 4 个示例数据库，以某种方式修改表中的一列或多列。

❖ **警告**　在修改版示例数据库中，所有的示例查询都会修改数据，因此有些查询仅在第一次运行时的效果与预期的相同。例如，如果你运行一个 UPDATE 查询来修改顾客姓名或保龄球队名称，而它使用 WHERE 子句来查找要修改的行，那么再次运行时，第一次运行时所做的修改将导致无法找到要修改的行。如果你要多次练习解决问题，可使用示例脚本或备份恢复数据库。

另外，如果你使用的是 MySQL，MySQL Workbench 的查询编辑器（Query Editor）默认处于安全更新模式，这要求在 WHERE 子句必须使用主键来指定查找条件。在启用了安全更新模式的情况下，这里的很多查询都不能运行。要禁用安全更新模式，请选择 Edit➢ Preferences➢SQL Editor，并取消选择底部附近的复选框 Safe Updates。

在每个示例中，我都显示了修改前后的目标表，指出了 UPDATE 语句修改了多少行。在修改的行数前面，指出了在本书配套网站提供的示例数据库中，保存查询时使用的名称。另外，对于每个 UPDATE 查询，我还编写了配套的 SELECT 查询（在 MySQL、PostgreSQL 和 Microsoft SQL Server 中存储为视图），你可使用它们来获悉哪些内容将被修改。给这些配套查询命名时，我在原始查询的名称后面添加了_Query。我将每个查询都存储到了相应的示例数据库中，名称以 CH15 打头。示例数据库可在本书配套网站中找到。你可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

❖ **说明**　别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录 B。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。这些示例都假定你深入研究并掌握了本书前面介绍的概念，尤其是连接和子查询。

所有的示例语句都有配套的 SELECT 语句，你可在运行 UPDATE 语句前使用它们来列出将受影响的行和修改后的值。这些查询(视图)与对应 UPDATE 语句同名，但加上了后缀_Query。例如，对于名为 CH15_Update_Order_Totals_Subquery 的 UPDATE 语句，与之配套的 SELECT 语句名为 CH15_Update_Order_Totals_Subquery。配套语句只指出受影响的行，而不修改任何数据。

- **Sales Orders 数据库**

"对于发货日期比下单日期晚 30 天以上的订单，将报价下调 2%。"

我们换种说话，让这个问题的措词更接近 SQL 语法。

“对于发货日期比下单日期晚 30 天以上的订单，修改订单详情，将报价设置为原报价乘以 0.98。”

转换/整理　Update ~~the~~ order details ~~table by~~ set~~ting the~~ quoted price ~~equal to~~ = ~~the~~ quoted price ~~times~~ * 0.98 where ~~the~~ order ID ~~is~~ in ~~the~~ (select~~ion of~~ order IDs from ~~the~~ orders ~~table~~ where ship date ~~minus~~ – order date ~~is greater than~~ > 30（在订单表中，从发货日期减去下单日期的结果大于 30 的行中选择订单 ID，生成一个订单 ID 列表；更新订单详情表，对于订单 ID 在上述列表中的行，将其中的报价设置为原报价乘以 0.98）

SQL

```
UPDATE Order_Details
SET QuotedPrice = QuotedPrice * 0.98
WHERE OrderID IN
    (SELECT OrderID
    FROM Orders
    WHERE (ShipDate - OrderDate) > 30)
```

❖ **说明**　这个解决方案假定你使用的数据库系统支持将两个日期相减，得到它们相隔的天数。有关这方面的细节，请参阅你使用的数据库系统的文档。

执行 UPDATE 查询前，CH15_Adjust_Late_Order_Prices_Query 返回的结果集

OrderNumber	ProductNumber	QuotedPrice	UpdatedPrice
291	1	$1,200.00	1176
291	14	$139.95	137.15
291	30	$43.65	42.78
371	9	$32.01	31.37
371	22	$79.54	77.95
371	35	$37.83	37.07
387	1	$1,200.00	1176
387	6	$635.00	622.3
	<< 其他行 >>		

执行 CH15_Adjust_Late_Order_Prices 后的 Order_Details 表（修改了 29 行）

OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
291	1	$1,176.00	4
291	14	$137.15	2
291	30	$42.78	6
371	9	$31.37	6
371	22	$77.95	5
371	35	$37.07	6
387	1	$1,176.00	4
387	6	$622.30	4
	<< 其他行 >>		

“确保所有自行车的零售价都比报价最低的供应商提供的批发价至少高 45%。”

换种说法，这样陈述这个问题：

“修改商品表，对于类别 ID 为 2 且零售价低于报价最低的供应商提供的批发价的 1.45 倍的商品，将其零售价设置为报价最低的供应商提供的批发价的 1.45 倍。”

转换/整理　Update ~~the~~ products ~~table by~~ set~~ting the~~ retail price ~~equal to~~ = 1.45 ~~times~~ * ~~the~~ (select~~ion of the unique~~ distinct wholesale price from ~~the~~ product vendors ~~table~~ where ~~the~~ product vendors ~~table's~~ product number ~~is equal to~~ = ~~the~~ products ~~table's~~ product number and ~~the~~ wholesale price ~~is equal to~~ = ~~the~~ (select~~ion of the~~ min~~imum~~ (wholesale price) from ~~the~~ product vendors ~~table~~ where ~~the~~ product vendors ~~table's~~ product number ~~is equal to~~ = ~~the~~ products ~~table's~~ product number)) where ~~the~~ retail price ~~is less than~~ < 1.45 times ~~the~~ (select~~ion of the unique~~ distinct wholesale price from ~~the~~ product vendors ~~table~~ where ~~the~~ product vendors

~~table's~~ product number ~~is equal to~~ = ~~the~~ products ~~table's~~ product number and ~~the~~ wholesale price ~~is equal to~~ = ~~the~~ (selecti~~on of the~~ min~~imum~~ (wholesale price) from ~~the~~ product vendors ~~table~~ where ~~the~~ product vendors ~~table's~~ product number ~~is equal to~~ = ~~the~~ products ~~table's~~ product number)) and ~~the~~ category ID ~~is equal to~~ = 2（更新商品表，对于类别 ID 为 2 且零售价低于报价最低的供应商提供的批发价的 1.45 倍的商品，将其零售价设置为报价最低的供应商提供的批发价的 1.45 倍。对于特定的商品，这样找出报价最低的供应商提供的批发价：在商品供应商表中，从商品编号与当前商品编号相同的行中选择批发价，再找出其中最低的批发价；在商品供应商表中，从商品编号与当前商品编号相同且批发价与前述最低批发价相同的行中选择批发价，并返回其中不同的批发价）

SQL

```
UPDATE Products
SET RetailPrice = ROUND(1.45 *
  (SELECT DISTINCT WholeSalePrice
   FROM Product_Vendors
   WHERE Product_Vendors.ProductNumber
    = Products.ProductNumber
    AND WholeSalePrice =
      (SELECT MIN(WholeSalePrice)
       FROM Product_Vendors
       WHERE Product_Vendors.ProductNumber
        = Products.ProductNumber)), 0)
WHERE RetailPrice < 1.45 *
    (SELECT DISTINCT WholeSalePrice
     FROM Product_Vendors
     WHERE Product_Vendors.ProductNumber
         = Products.ProductNumber
       AND WholeSalePrice =
         (SELECT MIN(WholeSalePrice)
          FROM Product_Vendors
          WHERE Product_Vendors.ProductNumber
                = Products.ProductNumber))
   AND CategoryID = 2
```

❖ **说明** 在 Microsoft Access 示例数据库中，这个问题是在 UPDATE 子句中使用连接来解决的，因为 Access 不支持在 SET 子句中使用包含关键字 DISTINCT 的子查询（它声明这样的查询是不可更新的，因为使用了 DISTINCT）。另外，这个解决方案将计算得到的价格取整为最接近的美元数（没有小数部分）。大多数商业实现支持函数 ROUND，虽然 SQL 标准没有明确地定义它。

我原本还可以使用一个子查询在 Categories 表中找出类别描述 Bikes 对应的类别 ID，但考虑到这个查询已经很复杂了，因此没有再添加另一个子查询。最后，我使用了 DISTINCT 来处理批发价，因为可能有多家供应商的批发价都是最低的，而我希望这个子查询只返回一个用于比较的值。

执行 UPDATE 查询前，CH15_Adjust_Bike_Retail_Price_Query 返回的结果集（1 行）

ProductNumber	ProductName	RetailPrice	UpdatedPrice
2	Eagle FS-3 Mountain Bike	$1,800.00	1840

执行 CH15_Adjust_Bike_Retail_Price 后的 Products 表（修改了 1 行）

ProductNumber	ProductName	RetailPrice
2	Eagle FS-3 Mountain Bike	$1,840.00

❖ **说明** 如果你查看 Product_Vendors 表中的所有自行车（它们的商品编号为 1、2、6 和 11），将发现只有 2 号商品当前的零售价低于报价最低的供应商提供的批发价的 1.45 倍。对于这种商品编号为 2 的自行车，供应商 ID 为 6 的供应商的批发价为 1269 美元，其 1.45 倍为 1840.05 美元，而这个查询将其取整到最接近的美元数。

- **Entertainment Agency 数据库**

“将所有经纪人的薪水都上调 6%。”

换种说法，这样陈述这个问题：

“修改经纪人表，将所有的薪水都上调 6%。”

转换/整理　Update ~~the~~ agents ~~table by~~ set~~ting~~ salary ~~equal to~~ = salary ~~times~~ * 1.06（更新经纪人表，将薪水设置为原来的 1.06 倍）

SQL

```
UPDATE Agents
SET Salary = ROUND(Salary * 1.06, 0)
```

❖ **说明**　这里也使用了大多数商业实现支持的函数 ROUND，并取整到不包含小数部分。有关你使用的数据库系统的取整详情，请参阅其文档。

执行 UPDATE 查询前，CH15_Give_Agents_6Percent_Raise_Query 返回的结果集（9 行）

AgentID	AgtFirstName	AgtLastName	Salary	NewSalary
1	William	Thompson	$35,000.00	37100
2	Scott	Bishop	$27,000.00	28620
3	Carol	Viescas	$30,000.00	31800
4	Karen	Smith	$22,000.00	23320
5	Marianne	Wier	$24,500.00	25970
6	John	Kennedy	$33,000.00	34980
7	Caleb	Viescas	$22,100.00	23426
8	Maria	Patterson	$30,000.00	31800
9	Daffy	Dumbwit	$50.00	53

执行 UPDATE 查询 CH15_Give_Agents_6Percent_Raise 后的 Agents 表（修改了 9 行）

AgentID	AgtFirstName	AgtLastName	Salary	AgentID
1	William	Thompson	$37,100.00	1
2	Scott	Bishop	$28,620.00	2
3	Carol	Viescas	$31,800.00	3
4	Karen	Smith	$23,320.00	4
5	Marianne	Wier	$25,970.00	5
6	John	Kennedy	$34,980.00	6
7	Caleb	Viescas	$23,426.00	7
8	Maria	Patterson	$31,800.00	8
9	Daffy	Dumbwit	$53.00	9

“更正演出合约的价格，将其设置为演唱组合每日费用与演出天数的乘积，再加上 15%的佣金。”

我们换种说法：

“修改演出合约表，将合约价格设置为演出天数与演唱组合每日费用的乘积的 1.15 倍。”

转换/整理　Update ~~the~~ engagements ~~table by~~ set~~ting the~~ contract price ~~equal to~~ = 1.15 ~~times~~ * ~~the~~ (end date ~~minus~~ – ~~the~~ start date ~~plus~~ + 1) ~~and then times the~~ * (select~~ion of the~~ entertainer price per day from ~~the~~ entertainers ~~table~~ where ~~the~~ entertainers ~~table~~ entertainer ID ~~is equal to~~ = ~~the~~ engagements ~~table~~ entertainer ID（更新演出合约表，设置合约价格：对于演出合约表中的每一行，从演唱组合表选择相应演唱组合 ID 对应的每日费用，将其乘以结束日期减去开始日期再加 1 的结果，再乘以 1.15，然后将合约价格设置为这样计算得到的结果）

SQL

```
UPDATE Engagements
SET Engagements.ContractPrice =
   ROUND(1.15 * (EndDate - StartDate + 1) *
   (SELECT EntPricePerDay
    FROM Entertainers
     WHERE Entertainers.EntertainerID =
       Engagements.EntertainerID), 0)
```

❖ **说明** 这个解决方案假定你使用的数据库系统支持将两个日期相减以得到它们相隔的天数。有关这方面的细节，请参阅你使用的数据库系统的文档。

为获得实际的天数，我将两个日期的差加上了1，因为在演出合约的第一天和最后一天都有演出。对于时间只有一天的演出合约，显然需要这样做：开始日期和结束日期相同，因此它们的差为零，但演出时间刚好一天。

小数部分为0.5时，Microsoft Access使用银行家取整法，结果为最接近的偶数。例如，3.5被取整到4，2.5被取整到2。这可能导致有些结果相差1美元。

演唱组合的每日费用

EntertainerID	EntStageName	EntPricePerDay
1001	Carol Peacock Trio	$175.00
1002	Topazz	$120.00
1003	JV & the Deep Six	$275.00
1004	Jim Glynn	$60.00
1005	Jazz Persuasion	$125.00
1006	Modern Dance	$250.00
1007	Coldwater Cattle Company	$275.00
1008	Country Feeling	$280.00
1009	Katherine Ehrlich	$145.00
1010	Saturday Revue	$250.00
1011	Julia Schnebly	$90.00
1012	Susan McLain	$75.00
1013	Caroline Coie Cuartet	$250.00

执行UPDATE查询前，CH15_Calculate_Entertainment_ContractPrice_Query返回的结果集（111行）

Engagement Number	ContractPrice	NewContractPrice	EntertainerID
2	$200.00	345	1004
3	$590.00	863	1005
4	$470.00	483	1004
5	$1,130.00	1265	1003
6	$2,300.00	1610	1008
7	$770.00	1104	1002
8	$1,850.00	2530	1007
9	$1,370.00	3163	1010
10	$3,650.00	3163	1003
	<< 其他行 >>		

执行CH15_Calculate_Entertainment_ContractPrice后的Engagements表（修改了111行）

EngagementNumber	ContractPrice	EntertainerID
2	$345.00	1004
3	$862.00	1005
4	$483.00	1004
5	$1,265.00	1003
6	$1,610.00	1008
7	$1,104.00	1002
8	$2,530.00	1007
9	$3,162.00	1010
10	$3,162.00	1003
	<< 其他行 >>	

❖ **说明**　在 Engagements 表中，原来的合约价格是一些位于合理范围内的随机值，这个合理范围是我在生成示例数据时选择的。显然，这个更新查询更正了每个值，使其更合理——基于演唱组合的每日费用。

- **School Scheduling 数据库**

“对于邮政编码为 98270 或 98271 的所有学生，将其区号改为 360。”

换种说法，这样陈述这个问题：

“修改学生表，将邮政编码为 98270 或 98271 的学生的区号都设置为 360。”

转换/整理　Update ~~the~~ students ~~table by~~ set~~ting the~~ area code ~~equal to~~ = ‘360’ where ~~the~~ student zip code ~~is~~ in ~~the list~~ (‘98270’, ~~and~~ ‘98271’)（更新学生表，对于邮政编码在列表(98270, 98271)中的学生，将其区号设置为 360）

SQL
```
UPDATE Students
SET Students.StudAreaCode = '360'
WHERE Students.StudZipCode IN ('98270', '98271')
```

执行 UPDATE 查询前，CH15_Fix_Student_AreaCode_Query 返回的结果集（2 行）

Student ID	Stud FirstName	Stud LastName	Stud ZipCode	Stud AreaCode	NewStud AreaCode
1007	Elizabeth	Hallmark	98271	253	360
1017	George	Chavez	98270	206	360

执行 CH15_Fix_Student_AreaCode 后的 Students 表（修改了 2 行）

StudentID	StudFirstName	StudLastName	StudZipCode	StudAreaCode
1001	Kerry	Patterson	78284	210
1007	Elizabeth	Hallmark	98271	360
1008	Sara	Sheskey	97208	503
		<< 其他行 >>		
1016	Steve	Pundt	75204	972
1017	George	Chavez	98270	360
1018	Richard	Lum	98115	206
1019	Daffy	Dumbwit	98002	425

“基于已结束的课程重新计算所有学生的平均成绩。”

换种说法，这样陈述这个问题：

“修改学生表，将平均成绩设置为学分与成绩的乘积总和除以学分总和。”

转换/整理　Update ~~the~~ students ~~table by~~ set~~ting the~~ student GPA ~~equal to~~ = ~~the~~ (selec~~tion of the~~ sum ~~of~~ (credits ~~times~~ * grade) ~~divided by~~ / ~~the~~ sum ~~of~~ (credits) from ~~the~~ classes ~~table~~ inner join~~ed with the~~ student schedules ~~table~~ on classes.class ID ~~in the classes table matches~~ = student_ schedules.class ID ~~in the student schedules table~~ where ~~the~~ class status ~~is~~ = ~~complete~~ 2 and ~~the~~ student schedules ~~table~~ student ID ~~is equal to~~ = ~~the~~ students ~~table~~ student ID)［更新学生表，将学生的平均成绩设置为按如下方法计算得到的值：基于课程 ID 相同将课程表内连接到学生选课表，选择课程状态为“已结束”（2）且学生 ID 与当前学生的 ID 相同的行，计算这些行的学分与成绩的乘积总和，再除以学分总和］

SQL
```
UPDATE Students SET Students.StudGPA =
    (SELECT ROUND (SUM(Classes.Credits *
        Student_Schedules.Grade) /
       SUM(Classes.Credits), 3)
     FROM Classes
     INNER JOIN Student_Schedules
     ON Classes.ClassID = Student_
         Schedules.ClassID
    WHERE (Student_Schedules.ClassStatus = 2)
     AND (Student_Schedules.StudentID =
        Students.StudentID))
```

执行 UPDATE 查询前，CH15_Update_Student_GPA_Query 返回的结果集（19 行）

StudentID	StudFirstName	StudLastName	StudGPA	NewStudGPA
1001	Kerry	Patterson	74.465	81.075
1002	David	Hamilton	78.755	80.09
1003	Betsy	Stadick	85.235	80.31
1004	Janice	Galvin	81	85.042
1005	Doris	Hartwig	72.225	85.135
1006	Scott	Bishop	88.5	77.512
1007	Elizabeth	Hallmark	87.65	72.098
1008	Sara	Sheskey	84.625	85.695
		<< 其他行 >>		

执行 CH15_Update_Student_GPA 后的 Students 表（修改了 19 行）

StudentID	StudFirstName	StudLastName	StudGPA
1001	Kerry	Patterson	81.075
1002	David	Hamilton	80.09
1003	Betsy	Stadick	80.31
1004	Janice	Galvin	85.042
1005	Doris	Hartwig	85.135
1006	Scott	Bishop	77.512
1007	Elizabeth	Hallmark	72.098
1008	Sara	Sheskey	85.695
		<< 其他行 >>	

❖ **说明**　Microsoft Access 不支持包含聚合函数的子查询，因此在 Microsoft Access 中解决这个问题时，调用了一系列内置函数，并使用了一个基于 Student_Schedules 表和 Classes 表的预定义视图。另外，如果你使用上述 SQL，将发现对于没有注册任何课程的学生（最后一个），结果为 Null。在全部 4 个示例数据库中，我都避免了出现 Null 值——使用各个数据库系统中提供的函数将其替换为 0。第 19 章将介绍如何使用 CASE 来避免这种问题。

- **Bowling League 数据库**

“计算并更新所有投球手的总得分、比赛局数、当前平均得分和当前让分数。”

❖ **说明**　在 13.8 节，你使用 SELECT 查询计算过让分数，请参阅示例数据库 Bowling League 中的查询 CH13_Bowler_Average_Handicap。别忘了，让分数为 200 减去投球手平均原始得分的 90%。

我们换种说法，这样陈述这个问题：

“根据投球手得分表计算总得分、比赛局数、平均得分和让分数，并相应地修改投球手表。”

转换/整理	Update ~~the~~ bowlers ~~table by~~ set~~ting the~~ total pins ~~equal to~~ = ~~the~~ (select~~ion of the~~ sum ~~of the~~ (raw score) from ~~the~~ bowler scores ~~table~~ where ~~the~~ bowler scores ~~table~~ bowler ID ~~is equal to~~ = ~~the~~ bowlers ~~table~~ bowler ID), ~~and the~~ games bowled ~~equal to~~ = ~~the~~ (select~~ion of the~~ count ~~of the~~ (raw score) from ~~the~~ bowler scores ~~table~~ where ~~the~~ bowler scores ~~table~~ bowler ID ~~is equal to~~ = ~~the~~ bowlers ~~table~~ bowler ID), ~~and the~~ current average ~~equal to~~ = ~~the~~ (select~~ion of the average~~ avg ~~of the~~ (raw score) from ~~the~~ bowler scores ~~table~~ where ~~the~~ bowler scores ~~table~~ bowler ID ~~is equal to~~ = ~~the~~ bowlers ~~table~~ bowler ID), ~~and the~~ current handicap ~~equal to~~ = ~~the~~ (select~~ion of~~ 0.9 ~~times~~ * (200 ~~minus~~ − ~~the average~~ avg ~~of the~~ (raw score)) from ~~the~~ bowler scores ~~table~~ where ~~the~~ bowler scores ~~table~~ bowler ID ~~is equal to~~ = ~~the~~ bowlers ~~table~~ bowler ID)（更新投球手表，将总得分设置为投球手得分表中投球手 ID 与当前投球手的 ID 相同的行中的原始得分总和，将比赛局数设置为投球手得分表中投球手 ID 与当前投球手的 ID 相同的行数，将平均得分设置为投球手得分表中投球手 ID 与当前投球手的 ID 相同的行中的原始得分平均值，将让分数设置为 200 减去投球手得分表中投球手 ID 与当前投球手的 ID 相同的行中的原始得分平均值的 90%）

SQL	

```
UPDATE Bowlers SET Bowlers.BowlerTotalPins =
    (SELECT SUM(RawScore)
     FROM Bowler_Scores
     WHERE Bowler_Scores.BowlerID =
         Bowlers.BowlerID),
      Bowlers.BowlerGamesBowled =
    (SELECT COUNT(Bowler_Scores.RawScore)
     FROM Bowler_Scores
     WHERE Bowler_Scores.BowlerID =
         Bowlers.BowlerID),
      Bowlers.BowlerCurrentAverage =
    (SELECT ROUND(AVG(Bowler_Scores.RawScore), 0)
     FROM Bowler_Scores
     WHERE Bowler_Scores.BowlerID =
           Bowlers.BowlerID),
      Bowlers.BowlerCurrentHcp =
    (SELECT ROUND(0.9 *
       (200 - ROUND(AVG(Bowler_Scores.RawScore),
       0)), 0)
     FROM Bowler_Scores
     WHERE Bowler_Scores.BowlerID =
         Bowlers.BowlerID)
```

执行 UPDATE 查询前，CH15_Calc_Bowler_Pins_Avg_Hcp_Query 返回的结果集（34 行）

Bowler ID	Bowler Total Pins	New Bowler Total Pins	Bowler Games Bowled	New Bowler Games Bowled	Bowler Current Average	New Bowler Current Average	Bowler Current Hcp	New Bowler Current Hcp
1	5790	6242	39	42	148	149	47	46
2	6152	6581	39	42	158	157	38	39
3	6435	6956	39	42	165	166	32	31
4	5534	5963	39	42	142	142	52	52
5	5819	6269	39	42	149	149	46	46
6	6150	6654	39	42	158	158	38	38
7	6607	7042	39	42	169	168	28	29
8	5558	5983	39	42	143	142	51	52
9	5874	6319	39	42	151	150	44	45
10	6184	6702	39	42	159	160	37	36
<< 其他行 >>								

执行 CH15_Calc_Bowler_Pins_Avg_Hcp 的 Bowlers 表（修改了 34 行）

BowlerID	<<other columns>>	Bowler TotalPins	Bowler GamesBowled	Bowler CurrentAverage	Bowler CurrentHcp
1	…	6242	42	149	46
2	…	6581	42	157	39
3	…	6956	42	166	31
4	…	5963	42	142	52
5	…	6269	42	149	46
6	…	6654	42	158	38
7	…	7042	42	168	29
8	…	5983	42	142	52
9	…	6319	42	150	45
10	…	6702	42	160	36
<< 其他行 >>					

> ❖ **说明** Microsoft Access 不支持包含聚合函数的子查询，因此在 Microsoft Access 中解决这个问题时，调用了一系列内置函数。另外，如果你使用上述 SQL，将发现对于没有参加任何比赛的投球手（最后两个），结果为 Null。在全部 4 个示例数据库中，我都避免了出现 Null 值——使用各个数据库系统中提供的函数将其替换为 0。第 19 章将介绍如何使用 CASE 来避免这种问题。

"将原来安排在保龄球馆 Sports World Lanes 举行的所有联赛的比赛地点都改为 Oasis Lanes。"

换种说法，这样陈述这个问题：

"修改联赛表，将原来安排在保龄球馆 Sports World Lanes 举行的所有联赛的比赛地点都改为 Oasis Lanes。"

转换/整理	Update ~~the~~ tournaments ~~table by~~ set~~ting the~~ tourney location ~~equal to~~ = 'Oasis Lanes' where ~~the original~~ tourney location ~~is equal to~~ = 'Sports World Lanes'（更新联赛表，将比赛地点为 Sports World Lanes 的联赛的比赛地点设置为 Oasis Lanes）

SQL	`UPDATE Tournaments`
	`SET TourneyLocation = 'Oasis Lanes'`
	`WHERE TourneyLocation = 'Sports World Lanes'`

执行 UPDATE 查询前，CH15_Change_Tourney_Location_Query 返回的结果集（3 行）

TourneyID	TourneyLocation	NewTourneyLocation
5	Sports World Lanes	Oasis Lanes
12	Sports World Lanes	Oasis Lanes
18	Sports World Lanes	Oasis Lanes

执行 CH15_Change_Tourney_Location 后的 Tournaments 表（修改了 3 行）

TourneyID	TourneyDate	TourneyLocation
	<< 其他行 >>	
5	2017-10-02	Oasis Lanes
	<< 其他行 >>	
12	2017-11-20	Oasis Lanes
	<< 其他行 >>	
18	2018-08-02	Oasis Lanes
19	2018-08-09	Imperial Lanes
20	2018-08-16	Totem Lanes

15.6 小结

本章首先简要地讨论了用于修改表中数据而不是获取数据的 UPDATE 语句，简要地介绍了 UPDATE 语句的语法，并介绍了一个简单的示例——使用表达式更新一个表的所有行中的一列。

接下来通过示例演示了如何使用 WHERE 子句来筛选出要更新的行，如何先编写 SELECT 查询来核实你将更新正确的行，以及如何将 SELECT 查询转换为所需的 UPDATE 语句。

接下来解释了事务的重要性，以及如何使用它们来防范错误，即确保要么执行所有的修改，要么不对表做任何修改。接着讨论了 UPDATE 语句，并演示了如何使用单个 UPDATE 查询同时更新一个表中的多列。

然后，介绍了如何在 UPDATE 查询中使用子查询，诠释了如何在 WHERE 子句中使用子查询来编写复杂的筛选器，并演示了如何在 SET 子句中使用子查询来生成要赋给列的新值。

最后列举了一些编写 UPDATE 查询的示例。

下一节列出了多个你可独自解决的问题。

15.7　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 对于 2017 年 10 月的消费总额超过 50 000 美元的顾客，给其所有订单都打 95 折。

  提示：需要在子查询中嵌套一个子查询，以获取 2017 年 10 月的消费总额超过 50 000 美元的所有顾客，进而获取这些顾客的所有订单的订单号。

  解决方案见 CH15_Give_Discount_To_Good_October_Customers（修改 639 行）。执行这个查询后，务必运行 CH15_Update_Order_Totals_Subquery 来更正 Orders 表中的订单总价。

  (2) 将配饰（1 类商品）的零售价设置为报价最高的供应商提供的批发价加 35%。

  提示：参阅 15.5 节的 CH15_Adjust_Bike_Retail_Price 了解解决方法。

  解决方案见 CH15_Adjust_Accessory_Retail_Price（修改了 11 行）。

- **Entertainment Agency 数据库**

  (1) 对于 2017 年 10 月的消费总额超过 3000 美元的顾客，给其所有演出合约的价格都打 98 折。

  提示：使用一个聚合子查询来找出 2017 年 10 月的消费总额的顾客。

  解决方案见 CH15_Discount_Good_Customers_October（修改了 34 行）。

  (2) 对于总销售额超过 20 000 美元的经纪人，将其佣金比例提高 0.5%。

  提示：使用一个聚合子查询来找出销售总额超过 20 000 美元的经纪人。

  解决方案见 CH15_Reward_Good_Agents（修改了 3 行）。

- **School Scheduling 数据库**

  (1) 将全职终身教员的薪水提高 5%。

  提示：在 WHERE 子句中使用一个子查询在教员表中找出匹配的教员 ID，即状态列全职、终身列为真（1 或–1，具体是哪个取决于你使用的数据库系统）行中的教员 ID。

  解决方案见 CH15_Give_FullTime_Tenured_Raise（修改了 21 行）。

  (2) 找出邮政编码为 98270 或 98271 的所有教员，将他们的区号都改为 360。

  解决方案见 CH15_Fix_Staff_AreaCode（修改了 2 行）。

- **Bowling League 数据库**

  (1) 将保龄球队 Huckleberrys 改名为 Manta Rays。

  解决方案见 CH15_Change_Huckleberry_Name（修改了 1 行）。

  (2) 根据每位投球手的邮政编码找出其所在的州和城市并更新这两项。

  提示：使用一个子查询从 WAZips 表中获取与邮政编码对应的城市，并使用一个查询获取对应的州。

  解决方案见 CH15_Update_Bowler_City_State（修改了 6 行）。

第 16 章 插入数据集

“小时候受到的教育让我相信，唯一值得做的事情就是为这世上正确的知识库添砖加瓦。”

——玛格丽特 · 米德

本章涵盖如下主题：

- 何谓插入
- INSERT 语句
- INSERT 的用途
- 语句举例
- 小结
- 练习

至此，你学习了如何创造性地从表中获取信息。前一章你还学习了如何使用 UPDATE 语句修改既有的数据。但数据最初是如何加入到表中的呢？它们当然不会自己进去！本章将告诉你答案——使用 INSERT 语句将行加入到表中。

16.1 何谓插入

大多数商业数据库系统自带了一个或多个图形界面程序，让你能够处理显示在屏幕上的数据。例如，在 Microsoft Office Access 中可打开任何表，为此只需找到相应的表对象并打开它，而 Access 将在数据表（datasheet）中显示表中的数据。数据表看起来像是由行和列组成的网格，你可滚动到数据表末尾并找到空白行，再在该行的列中输入数据，然后移到下一行并在表中插入一个新行。在 Access 中，还可以以同样的方式处理 Microsoft SQL Server 中的表。在 MySQL 查询浏览器中可执行类似的操作，而 Microsoft SQL Server、IBM DB2 和 Oracle 都提供了类似的工具。另外，你还可从第三方提供商（如 SQL Maestro）购买 PostgreSQL 图形设计工具。

当你输入新数据并让系统保存它们时，会发生什么呢？图形界面工具将执行一个命令，该命令使用 SQL 将你输入的数据加入到表中。这些程序使用的是 SQL INSERT 语句。如果你浏览示例文件，将在其中找到我生成的用于将数据加入示例数据库的脚本。例如，文件 01 EntertainmentAgencyData.SQL 的开头几行类似于下面这样：

```
USE EntertainmentAgencyExample;
INSERT INTO Customers
  (CustomerID, CustFirstName, CustLastName, CustStreetAddress,
   CustCity, CustState, CustZipCode, CustPhoneNumber)
  VALUES (10001, 'Doris', 'Hartwig', '4726 - 11th Ave. N.E. ',
   'Seattle' 'WA', '98105', '555-2671');
INSERT INTO Customers
  (CustomerID, CustFirstName, CustLastName, CustStreetAddress,
   CustCity, CustState, CustZipCode, CustPhoneNumber)
  VALUES (10002, 'Deb', 'Waldal', '908 W. Capital Way',
   'Tacoma', 'WA', '98413', '555-2496');
```

其中的第一个命令（USE）告诉数据库系统，接下来的命令将使用哪个数据库。每条 INSERT 语句都让数据库系

统将一行添加到指定的表中。看起来将数前条记录加载到示例数据库中是个烦琐的过程，但你将发现，每个加载数据的脚本只需几秒钟就能运行完毕。对于有些较简单的表，我使用图形用户界面直接将数据输入其中。为了给其他示例表生成数据，我编写了一些创建并执行 INSERT 语句的代码。如果你熟悉 Microsoft Office Access 和 Visual Basic，可在示例数据库 Sales Orders 的窗体 zfrmSellProducts 中找到生成示例数据的代码。

编写应用程序（无论是桌面应用程序还是 Web 应用程序）时，你通过编写代码在用户输入新数据时生成并执行合适的 INSERT 语句。在大多数情况下，你将使用形如 INSERT . . . VALUES 的 INSERT 语句来添加数据，但本章也将介绍另一种形式的 INSERT 语句，它让你能够轻松地在表之间复制数据。

❖ **说明**　所有示例语句和解决方案都可在相应示例数据库的“修改”版本——Sales Orders Modify、Entertainment Agency Modify、School Scheduling Modify 和 Bowling League Modify 中找到。

16.2　INSERT 语句

SQL 有两个主要的 INSERT 语句版本。在第一个版本中，你指定关键字 VALUES，并列出一系列值，让数据库系统将它们作为一个新行插入到指定的目标表中；第二个版本让你能够使用 SELECT 子句从表中获取要插入到目标表中的数据。我们先来看使用关键字 VALUES 的版本。

16.2.1　插入值

虽然 SQL 主要是为处理数据集而设计的，但在大多数情况下，你使用 INSERT 将单行输入添加到表中。要在表中添加一行，最简单的方式是使用包含 VALUES 子句的 INSERT 语句。图 16-1 是这种语句的语法图。

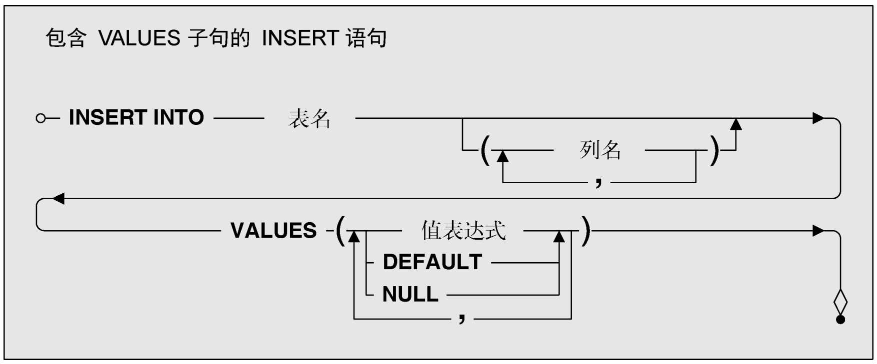


图 16-1　使用 VALUES 子句的 INSERT 语句的语法图

如你所见，这种语句以关键字 INSERT INTO 打头，接下来是要在其中添加行的表。如果你按列在表中被定义的顺序给所有列提供值，可省略列名列表（例如，在我用来加载示例数据的 INSERT 语句中，原本可以不指定列名列表，因为我给每列都提供了一个值）。然而，即便打算给所有列提供值，也建议你指定一个列表，在其中包含你要提供数据值的每个列。否则，如果有人在表定义中添加了列或修改了表定义中列的排列顺序，你编写的查询将不管用。要指定列名列表，可依次输入左括号、用逗号分隔的列名和右括号。

❖ **说明**　SQL 标准指出，表名也可以是视图名，但指定的视图必须是可更新、可插入的。很多数据库系统都支持在视图中插入行；在可更新或可插入的视图必须满足什么条件方面，每个数据库系统都有不同的规定。

在大多数情况下，如果使用了关键字 DISTINCT 或某个输出列是由表达式或聚合函数生成的，视图就是不可插入的。有些数据库系统还支持在定义视图时使用关键字 JOIN 和 ON 而不是表名。有关这方面的细节，请参阅你使用的数据库系统的文档。在本章的每条 INSERT 语句中，都只将单个表作为插入目标。

最后，指定关键字 VALUES、一个左括号、一个用逗号分隔的值表达式列表和一个右括号。请注意，值的排列顺序必须与列表中列的排列顺序一致；也就是说，第一个值表达式给列表中的第一列提供值，第二个值表达式给列表中的第二列提供值，以此类推。如果你给所有列都指定了值，但没有指定列名列表，则值的排列顺序必须与表定义中列的排列顺序相同。要让数据库系统使用为列定义的默认值，可使用关键字 DEFAULT，但如果没有定义默认值，这样做将导致错误。要提供 Null 值，可使用关键字 NULL，但如果列被定义为不接受 Null 值，这样做将导致错误。

本书前面说过，值表达式可以非常复杂，甚至可包含从目标表或其他表获取单个值的子查询。为了帮助你复习一下，图 16-2 显示了值表达式的语法图。

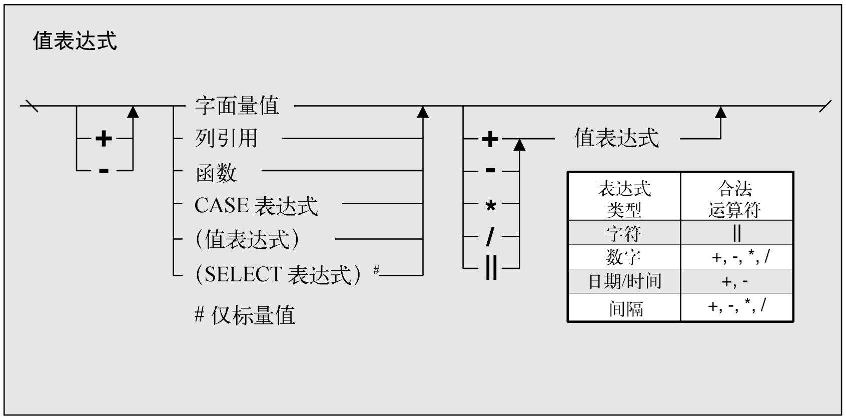


图 16-2　在 VALUES 子句中使用值表达式（其语法如这里所示）给目标表中的列指定值

❖ **说明**　并非在所有的数据库系统中，都可以在 INSERT 语句的 VALUES 子句中使用 SELECT 表达式或 CASE 表达式，详情请参阅你使用的数据库系统的文档。

下面来看看如何在示例数据库 Sales Orders 的 Employees 表中添加一行。与所有查询一样，要编写 INSERT 查询，必须知道表的结构。图 16-3 显示了 Employees 表的结构。

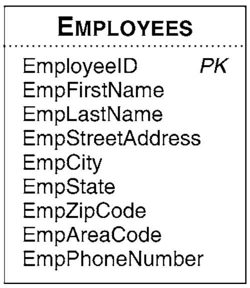


图 16-3　示例数据库 Sales Orders 中的 Employees 表

下面来编写一个查询。

❖ **说明**　贯穿本章都将采用第 4 章介绍的“请求/转换/整理/SQL”流程。

“添加一名新员工，她名为 Susan Metters，地址为‘16547 NE 132nd St, Woodinville, WA 98072’，区号为 425，电话号码为 555-7825。”

在最初的请求中，你通常不会列出涉及的列，但转换时必须考虑这一点。你可能这样转换这个添加一名新员工的请求：

转换	Insert into the employees table in the columns first name, last name, street address, city, state, ZIP Code, area code, and phone number the values Susan, Metters, 16547 NE 132nd St, Woodinville, WA, 98072, 425, and 555-7825（在员工表中的名、姓、街道地址、城市、州、邮政编码、区号和电话号码列中，分别插入值 Susan、Metters、16547 NE 132nd St、Woodinville、WA、98072、425 和 555-7825）
整理	Insert into ~~the~~ employees ~~table in the columns~~ (first name, last name, street address, city, state, ZIP Code, area code, ~~and~~ phone number) ~~the~~ values (‘Susan’, ‘Metters’, ‘16547 NE 132nd St’, ‘Woodinville’, ‘WA’, ‘98072’, 425, ~~and~~ ‘555-7825’)

SQL
```
INSERT INTO Employees
   (EmpFirstName, EmpLastName,
    EmpStreetAddress, EmpCity, EmpState,
    EmpZipCode, EmpAreaCode, EmpPhoneNumber)
VALUES ('Susan', 'Metters',
    '16547 NE 132nd St', 'Woodinville', 'WA',
    '98072', 425, '555-7825')
```

可在示例数据库 Sales Orders 的修改版中找到这个查询，它名为 CH16_Add_Employee。

你可能会问：为何没有给 Employees 表中的主键（EmployeeID）指定值呢？如果你有这样的疑问，请接着往下读！

16.2.2　生成下一个主键值

在前一节的示例查询中，我没有给主键（EmployeeID）指定值。在所有的数据库中，主键都必须有值。既然如此，这个查询会失败吗？

答案是否定的，但原因是我利用了几乎所有商业数据库系统都有的一个特殊功能。如果不在乎主键的值是多少（只要是独一无二的），通常可将主键定义为一种特殊的数据类型，这样每当你输入一个新行时，数据库系统都会将主键值加 1。在 Microsoft Access 中，这种特殊数据类型为 AutoNumber（实际上是具有特殊属性的整数）；在 Microsoft SQL Server 和 IBM DB2 中，这种特殊数据类型是 Identity（也是整数）；在 PostgreSQL 中，是序列（serial）数据类型（也是整数）；在 MySQL 中，可将数据类型设置为具有特殊属性 AUTO_INCREMENT 的整数。前述示例使用的 SQL 语法在全部 3 类示例数据库中都可行，因为在修改版示例数据库的几乎所有表中，主键列都被设置为特殊数据类型。

Oracle 数据库系统略有不同，它没有提供特殊数据类型，而让你定义 Sequence 伪列，并在需要给新行指定独一无二的值时引用伪列的 NEXTVAL 属性。在 Oracle 中，如果将 EmpID 定义成了伪列，就可像下面这样编写 SQL：

SQL
```
INSERT INTO Employees (EmployeeID, EmpFirstName,
   EmpLastName, EmpStreetAddress, EmpCity,
   EmpState, EmpZipCode, EmpAreaCode,
   EmpPhoneNumber)
VALUES (EmpID.NEXTVAL, 'Susan', 'Metters',
   '16547 NE 132nd St', 'Woodinville', 'WA',
   '98072', 425, '555-7825')
```

请注意，这里按列在表定义中的排列顺序给每列提供了一个值，但可以省略可选的列名列表，像下面这样编写 SQL：

SQL
```
INSERT INTO Employees
VALUES (EmpID.NEXTVAL, 'Susan', 'Metters',
   '16547 NE 132nd St', 'Woodinville', 'WA',
   '98072', 425, '555-7825')
```

前面说过，建议不要省略列名列表，因为如果这样做，当数据库管理员添加了列或调整了列定义的排列顺序后，查询将失败。这里呈现这种做法只是出于完整性考虑。

如果你认真思考了，可能会问：可不可以使用子查询表达式来生成下一个主键值呢？ SQL 标准确实支持这样做。你可以像下面这样编写 SQL：

```
SQL         INSERT INTO Employees (EmployeeID, EmpFirstName,
               EmpLastName, EmpStreetAddress, EmpCity,
               EmpState, EmpZipCode, EmpAreaCode,
               EmpPhoneNumber)
            VALUES ((SELECT MAX(EmployeeID)
                     FROM Employees) + 1, 'Susan', 'Metters',
               '16547 NE 132nd St', 'Woodinville', 'WA',
               '98072', 425, '555-7825')
```

但多个主要的数据库系统都不支持在 VALUES 子句中使用子查询，有关这方面的详情，请参阅你使用的数据库系统的文档。

❖ **说明** 在 Microsoft SQL Server 中，要在 IDENTITY 列中插入值，必须先执行如下命令：

```
SET IDENTITY_INSERT <table_name> ON;
```

执行完 INSERT 语句后，务必将这个选项设置为 OFF。

在 Microsoft Access、Microsoft SQL Server 或 MySQL 中，当你显式地在自动递增列中插入值后，这些数据库系统将自动更新下一个值。但 PostgreSQL 不会这样做。在 PostgreSQL 中，如果你在插入行时给 SERIAL 列指定了值，则必须像下面这样重置最后一个值：

```
SELECT setval('tablename_serialcolname_seq', <number>)
```

其中 tablename 是目标表的名称，serialcolname 是该表中 SERIAL 列的名称，而<number>是你提供的最后一个值。

16.2.3 使用 SELECT 插入数据

前面介绍的都是每次插入一行，因此你可能会问：为何将本章命名为“插入数据集”呢？从某种意义上说，一行中多列的值构成了一个数据集，但你可能认为集合就是多行。别担心，你也可插入多行，为此只需将 VALUES 子句替换为 SELECT 表达式。SELECT 表达式从一个或多个表中获取行，因此可将包含 SELECT 的 INSERT 语句视为一种强大的数据复制方式。图 16-4 显示了使用 SELECT 表达式的 INSERT 语句的语法图。

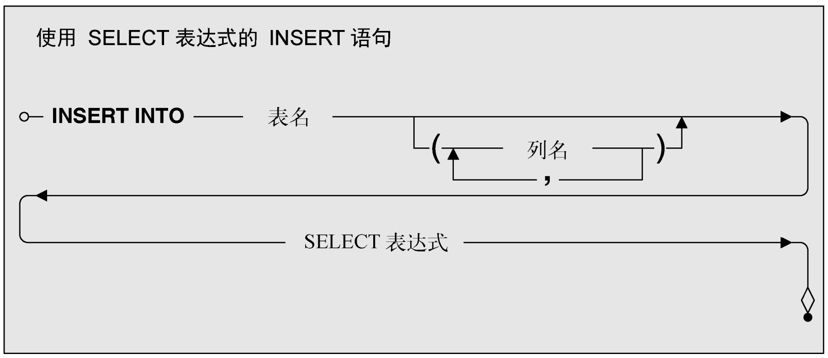


图 16-4 使用 SELECT 表达式的 INSERT 语句的语法图

请注意，在这个版本的 INSERT 语句中，开头没什么不同：在关键字 INSERT INTO 后面，指定目标表的表名。如果 SELECT 表达式返回的列的数量和排列顺序与目标表相同，可省略可选的列名列表，但前面说过，即便打算给所有的列提供值，也建议你包含列名列表，指出要给哪些列指定值。否则，如果有人在表定义中添加了列或修改了列的排列顺序，你编写的查询将失败。

如果你查看附录 A 的 SQL 语法图，将发现 SELECT 表达式不过是一条 SELECT 语句，但可通过 UNION、INTERSECT 或 EXCEPT 运算与其他 SELECT 语句组合起来（有关这 3 种运算的介绍，请参阅第 7 章；有关 UNION

的详细讨论，请参阅第 10 章）。图 16-5 显示了 SELECT 语句的语法图。

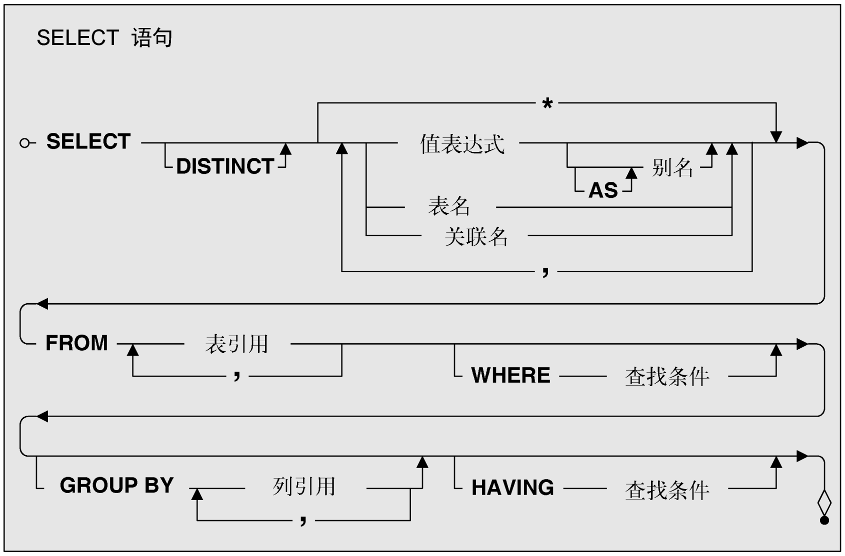


图 16-5　SELECT 语句的语法图

你可能还记得，本书前面说过，表引用可以是单个表名、一系列用逗号分隔的表名或复杂的多表连接；查找条件可以是简单的列与值的比较，使用 BETWEEN、IN、LIKE 或 NULL 的复杂测试，或使用子查询的极其复杂的谓词。简而言之，你可使用本书前面介绍的 SELECT 查询的所有强大功能，来指定要复制到表中的行集。

下面介绍使用包含 SELECT 表达式的 INSERT 语句可解决的一些问题。先来看一个简单的问题，它要求将一个表中的一行数据复制到另一个表中。

“我们刚聘用顾客 David Smith，需要将 Customers 表中有关他的详细信息复制到 Employees 表中。”

与编写其他查询一样，你需要熟悉涉及的表的结构。图 16-6 显示了这两个表的设计。

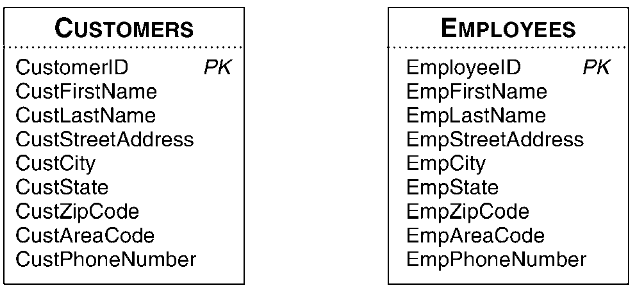


图 16-6　示例数据库 Sales Orders 中的 Customers 表和 Employees 表

我们以不同的方式陈述这个问题，以便更轻松地将其转换为 INSERT 查询：

“将 Customers 表中与顾客 David Smith 相关的列复制到 Employees 表中。”

转换	Insert into the employees table in the columns first name, last name, street address, city, state, ZIP code, area code, and phone number the selection of the first name, last name, street address, city, state, ZIP code, area code, and phone number columns from the customers table where the customer first name is ‘David’ and the customer last name is ‘Smith’（在顾客表中，从顾客名为 David、顾客姓为 Smith 的行中选择名、姓、街道地址、城市、州、邮政编码、区号和电话号码列；将它们分别插入到员工表的名、姓、街道地址、城市、州、邮政编码、区号和电话号码列中）

整理

Insert into ~~the~~ employees ~~table in the columns~~ (first name, last name, street address, city, state, ZIP code, area code, ~~and~~ phone number) ~~the~~ (select~~ion of the~~ first name, last name, street address, city, state, ZIP code, area code, ~~and~~ phone number ~~columns~~ from ~~the~~ customers ~~table~~ where ~~the~~ customer first name ~~is~~ = ‘David’ and ~~the~~ customer last name ~~is~~ = ‘Smith’)

SQL

```
INSERT INTO Employees
   (EmpFirstName, EmpLastName, EmpStreetAddress,
       EmpCity, EmpState, EmpZipCode,
       EmpAreaCode, EmpPhoneNumber)
   SELECT Customers.CustFirstName,
       Customers.CustLastName,
       Customers.CustStreetAddress,
       Customers.CustCity,
       Customers.CustState, Customers.CustZipCode,
       Customers.CustAreaCode,
       Customers.CustPhoneNumber

FROM Customers
WHERE (Customers.CustFirstName = 'David')
 AND (Customers.CustLastName = 'Smith')
```

请注意，我没有包含 EmployeeID 列，因为我要让数据库系统为插入的新行生成下一个独立无二的值。我将这个查询保存到了示例数据库 Sales Orders 的修改版中，并将其命名为 CH16_Copy_Customer_To_Employee。

由于只有一个姓名为 David Smith 的顾客，这个查询只将一行复制到 Employees 表中。插入的依然不是行集，但如你所见，使用 SELECT 表达式从另一个表中获取要插入的值很容易。

我们再来看一个问题，它可能需要插入数百行。在每个需要不断收集新信息的数据库应用程序中，都可能需要提供让用户能够归档的功能，即将某个时间点之前发生的所有交易都复制到备份表中。这样做旨在避免旧的历史数据降低新数据的处理速度：应用程序需要处理数千行数据，而这些数据表示的是很久以前发生的交易。

因此，你可能想编写一条 INSERT 语句，将特定日期之前发生的交易复制到专门存储历史数据的表中（有关如何从活动表中删除已归档的交易数据，请参阅下一章）。典型的请求可能类似于下面这样：

“将 2018 年 1 月 1 日之前的所有演出合约归档。”

在这里，Engagements 表和 Engagements_Archive 表的结构相同，如图 16-7 所示。

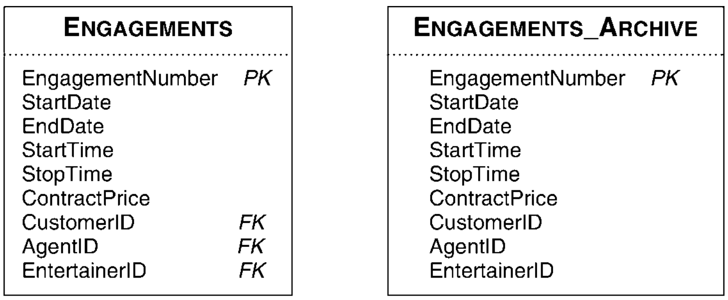


图 16-7 示例数据库 Entertainment Agency 中的 Engagements 表和 Engagements_Archive 表

在这个示例中，完全可以省略列名列表。转换起来非常容易，如下所示。

转换

Insert into the engagements archive table the selection of all columns from the engagements table where the engagement end date is earlier than January 1, 2018（在演出合约表中，从结束日期早于 2018 年 1 月 1 日的行中选择所有的列，并将它们插入到演出合约归档表中）

整理

Insert into ~~the~~ engagements archive ~~table the~~ select~~ion of all columns~~ * from ~~the~~ engagements ~~table~~ where ~~the~~ engagement end date ~~is earlier than~~ < ~~January 1, 2018~~ ‘2018-01-01’

SQL

```
INSERT INTO Engagements_Archive
   SELECT Engagements.*
   FROM Engagements
   WHERE Engagements.EndDate
      < '2018-01-01'
```

这太容易了，不是吗？但别忘了，我建议总是显式地列出列名，这样即便有人在其中一个表中添加了新列或调整了列的排列顺序，你编写的查询依然能够运行。建议像下面这样编写解决这个问题的 SQL，虽然这样做要多花些功夫。

SQL

```
INSERT INTO Engagements_Archive
    (EngagementNumber, StartDate, EndDate,
     StartTime, StopTime, ContractPrice,
     CustomerID, AgentID, EntertainerID)
    SELECT Engagements.EngagementNumber,
       Engagements.StartDate, Engagements.EndDate,
       Engagements.StartTime, Engagements.StopTime,
       Engagements.ContractPrice,
       Engagements.CustomerID,
       Engagements.AgentID,
Engagements.EntertainerID FROM Engagements
   WHERE Engagements.EndDate < '2018-01-01'
```

这个查询保存到了示例数据库 Entertainment Agency 的修改版中，名为 CH16_Archive_Engagements。

来看一种创造性地使用 SELECT 表达式的方式。请看下面的请求：

> “添加一种新商品，它名为 Hot Dog Spinner，零售价为 895 美元，属于自行车（Bikes）类别。”

图 16-8 列出了需要用到的表。

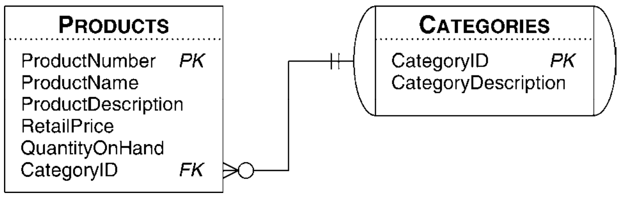


图 16-8　数据库 Sales Orders 中的 Products 表和相关的 Categories 表

显然，目标表是 Products，但其 CategoryID 列必须是数字值。前面的请求说“属于自行车类别”，那么如何找出要在 Products 表中插入的 CategoryID 值呢？使用 SELECT 表达式！你还需要给 ProductName 和 RetailPrice 列提供值，但别忘了在 SELECT 语句中，可给部分或所有输出列指定字面量值，因此你可从 Categories 表中获取相关的类别 ID，并以字面量值的方式提供其他要插入的值。我们以不同的方式陈述这个问题，再解决它（这个查询保存在了示例数据库 Sales Orders Modify 中，名为 CH16_Add_Product）。

> “在商品表中添加一行，其商品名为 Hot Dog Spinner，零售价为 895 美元，类别 ID 为类别表中类别描述 Bikes 对应的类别 ID。”

转换	Insert into the products table in the columns product name, retail price, and category ID the selection of ‘Hot Dog Spinner’ as the product name, 895 as the retail price, and category ID from the categories table where the category description is equal to ‘Bikes’ ［从类别表中选择 Hot Dog Spinner（作为商品名）、895（作为零售价）以及类别描述 Bikes 对应的类别 ID，将它们分别插入到商品表的商品名、零售价和类别 ID 列中］
整理	Insert into ~~the~~ products ~~table in the columns~~ (product name, retail price, ~~and~~ category ID) ~~the~~ selection~~ of~~ ‘Hot Dog Spinner’ as ~~the~~ product name, 895 as ~~the~~ retail price, ~~and~~ category ID from ~~the~~ categories ~~table~~ where ~~the~~ category description ~~is equal to~~ = ‘Bikes’

SQL

```
INSERT INTO Products
   (ProductName, RetailPrice, CategoryID)
SELECT 'Hot Dog Spinner' AS ProductName,
   895 AS RetailPrice, CategoryID
FROM Categories
WHERE CategoryDescription = 'Bikes'
```

你可能认为，使用 SELECT 表达式只能复制整行，但如你刚才所见，使用它也可从表中获取特定的值。在 16.4 一节，你将发现这种方法的一些有趣的用途。

16.3 INSERT 的用途

至此，你应对如何使用简单的VALUES子句或SELECT表达式插入一行或多行有了深入的认识。为了展示INSERT语句的用途有多广泛，最佳的方式是列举一些使用它可解决的问题，并提供一系列使用示例（参见16.4节）。下面是使用INSERT可解决的问题类型的很小一部分。

“通过复制表示Neil Patterson的记录，新建一条表示Matthew Patterson的投球手记录。”

“在Entertainment数据库中，新建一条表示顾客Kendra Hernandez的记录，其街道地址、城市、州、邮政编码和电话号码分别为457 211th St NE、Bothell、WA、98200和555-3945。”

“在数据库Sales Order中，新建一条表示顾客Mary Baker的记录，其街道地址、城市、州、邮政编码、区号和电话号码分别为7834 W 32nd Ct、Bothell、WA、98011、425和555-9876。”

“新建一个名为Italian的科目类别，其科目编码为ITA，属于人类学（Humanities）系。”

“新增一个名为Aardvarks的球队，该球队没有指定队长。”

“新增一个演出合约，顾客为Matt Berg，演出组合为Jazz Persuasion，演出时间为2018年8月15日和16日的晚上7点到11点，经纪人为Karen Smith。”

“新增一家供应商，其名称为Hot Dog Bikes，街道地址为1234 Main Street，所在城市为伊利诺伊州（IL）芝加哥（Chicago），邮政编码为 60620，电话号码为（773）555-6543，传真为（773）555-6542，网站地址为http://www.hotdogbikes.com/，电子邮件地址为Sales@hotdogbikes.com。”

“新增一门课程，其科目ID为4（Intermediate Accounting），教室为3315，学分5个，从2018年1月16日开始周二和周四上课，开始时间为下午3点，时长80分钟。”

“对于2017年举行的所有比赛场次，将其联赛ID、场次、局次和投球手得分归档。”

“新增音乐风格New Age。”

“将2018年1月1日前的所有订单及其订单详情归档。”

“Angel Kennedy想注册成为学生。她丈夫John已注册，请使用John的信息新建一条表示Angel的学生记录。”

“新建2019年的联赛和联赛场次，方法是复制2017年对应周的联赛和联赛场次。”

“经纪人Marianne Wier想与演唱组合签订演出合约，因此需要新建一条顾客记录，方法是复制经纪人表中相关列中的信息。”

“顾客Liz Keyser想再次订购他在2017年12月11日订购的商品；下单日期为2018年6月12日，送货日期为2018年6月15日。”

“教员Tim Smith想注册成为学生，请根据他的教员记录新建一条学生记录。”

“顾客Doris Hartwig想再次与一个演唱组合签约，她请该演唱组合在2017年12月1日演出过，但这次的演出日期为2018年8月1日。”

“顾客Angel Kennedy想再次订购她在2017年11月订购的所有商品；下单日期为2018年6月15日，送货日期为2018年6月18日。”

16.4 语句举例

知道了编写INSERT查询的机制后，来看一系列示例，它们都要求在表中添加一行或多行，且都来自4个示例数据库。

❖ **警告** 在修改版示例数据库中，所有的示例查询都会修改数据，因此有些查询仅在第一次运行时的效果与预期相同。例如，如果你运行一个INSERT查询来归档订单，而它使用了WHERE子句来查找要复制的行，那么再次运行时将失败，因为试图在归档表中插入重复的主键值。如果你要多次练习解决问题，可使用示例脚本或备份恢复数据库。

在每个示例中，都列出了示例 INSERT 语句插入的数据，指出了它插入了多少行。在行数前面，指出了在本书配套网站提供的示例数据库中，保存查询时使用的名称。另外，对于每个 INSERT 查询，我还编写了配套的 SELECT 查询（在 MySQL 和 Microsoft SQL Server 中存储为视图），你可使用它们来获悉将添加哪些内容。给这些配套查询命名时，在 INSERT 查询的名称后面添加了_Query。我将每个查询都存储到了相应的示例数据库中，名称以 CH16 打头。示例数据库可在本书配套网站中找到。你可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明**　别忘了，这些示例中使用的所有列名和表名都来自示例数据库，有关这些数据库的结构，请参阅附录 B。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。这些示例都假定你深入研究并掌握了本书前面介绍的概念。

- **Sales Orders 数据库**

“新增一家供应商，其名称为 Hot Dog Bikes，街道地址为 1234 Main Street，所在城市为伊利诺伊州（IL）芝加哥（Chicago），邮政编码为 60620，电话号码为(773)555-6543，传真为(773)555-6542，网站地址为 http://www.hotdogbikes.com/，电子邮件地址为 Sales@hotdogbikes.com。”

转换/整理	Insert into ~~the~~ vendors ~~table in the columns~~ (VendName, VendStreetAddress, VendCity, VendState, VendZipCode, VendPhoneNumber, VendFaxNumber, VendWebPage, ~~and~~ VendEMailAddress) ~~the~~ values (‘Hot Dog Bikes’, ‘1234 Main Street’, ‘Chicago’, ‘IL’, ‘60620’, ‘(773) 555-6543’, ‘(773) 555-6542’, ‘http://www.hotdogbikes.com/’, and ‘Sales@ hotdogbikes.com’)（在供应商表的 VendName、VendStreetAddress、VendCity、VendState、VendZipCode、VendPhoneNumber、VendFaxNumber、VendWebPage 和 VendEMailAddress 列中分别插入值 Hot Dog Bikes、1234 Main Street、Chicago、IL、60620, (773)555-6543、(773)555-6542、http://www.hotdogbikes.com/和 Sales@ hotdogbikes.com）

SQL

```sql
INSERT INTO Vendors
   (VendName, VendStreetAddress, VendCity,
    VendState, VendZipCode, VendPhoneNumber,
    VendFaxNumber, VendWebPage,
    VendEMailAddress)
VALUES ('Hot Dog Bikes', '1234 Main Street',
    'Chicago', 'IL', '60620', '(773) 555-6543',
    '(773) 555-6542', 'http://www.hotdogbikes.com/',
    'Sales@hotdogbikes.com')
```

CH16_Add_Vendor 在 Vendors 表中插入的行（1 行）

Vend Name	Vend Street Address	Vend City	Vend State	Vend ZipCode	Vend Phone Number	Vend Fax Number	Vend Web Page	Vend EMail Address
Hot Dog Bikes	1234 Main Street	Chicago	IL	60620	(773) 555-6543	(773) 555-6542	http://www .hotdogbikes .com/	Sales@ hotdogbikes. com

“将 2018 年 1 月 1 日前的所有订单及其订单详情归档。”

> ❖ **说明**　要将所有有关订单的信息归档，需要从 Orders 和 Order_Details 表中复制数据，因此需要两个查询。务必先运行复制 Orders 表中数据的 INSERT 查询，因为在 Orders_Details_Archive 表中，OrderID 列中的外键指向 Orders_Archive 表中的 OrderID 列。
>
> 如果你使用的数据库系统支持事务（参见第 15 章的讨论），可发起事务，依次运行复制订单的查询和复制订单详情的查询，再在这两个 INSERT 查询都没有出现错误时提交它们。如果第二个查询出错，可回滚事务，确保不复制任何订单行。复制有关订单的部分信息毫无意义。
>
> 由于要根据日期归档行，归档订单详情的查询必须使用一个子查询从 Orders 表中筛选出早于指定日期的订单号。

第 1 个转换/整理

Insert into ~~the~~ orders archive ~~table the~~ select~~ion of~~ order number, order date, ship date, customer ID, employee ID, ~~and~~ order total from ~~the~~ orders ~~table~~ where ~~the~~ order date ~~is earlier than~~ < '2018-01-01'（在订单表中，选择下单日期早于 2018-01-01 的订单的订单号、下单日期、发货日期、顾客 ID、员工 ID 和订单总价，将它们插入到订单归档表中）

SQL

```
INSERT INTO Orders_Archive
   SELECT OrderNumber, OrderDate, ShipDate,
       CustomerID, EmployeeID, OrderTotal
   FROM Orders
   WHERE OrderDate < '2018-01-01'
```

H16_Archive_2017_Orders 在 Orders_Archive 表中插入的行（添加了 594 行）

Order Number	OrderDate	ShipDate	CustomerID	EmployeeID	OrderTotal
1	2017-09-02	2017-09-05	1018	707	$12,751.85
2	2017-09-02	2017-09-04	1001	703	$816.00
3	2017-09-02	2017-09-05	1002	707	$11,912.45
4	2017-09-02	2017-09-04	1009	703	$6,601.73
5	2017-09-02	2017-09-02	1024	708	$5,544.75
6	2017-09-02	2017-09-06	1014	702	$9,820.29
7	2017-09-02	2017-09-05	1001	708	$467.85
8	2017-09-02	2017-09-02	1003	703	$1,492.60
9	2017-09-02	2017-09-05	1007	708	$69.00
10	2017-09-02	2017-09-05	1012	701	$2,607.00
<< 其他行 >>					

第 2 个转换/整理

Insert into ~~the~~ order details archive ~~table the~~ select~~ion of~~ order number, product number, quoted price, ~~and~~ quantity ordered from ~~the~~ order details ~~table~~ where ~~the~~ order number ~~is~~ in ~~the~~ (select~~ion of the~~ order number from ~~the~~ orders ~~table~~ where ~~the~~ order date ~~is earlier than~~ < '2018-01-01')（在订单表中，选择下单日期早于 2018-01-01 的订单的订单号，生成一个订单号列表；在订单详情表中，从订单号位于上述列表中的行中选择订单号、商品编号、报价和订购数量，并将它们插入到订单详情归档表中）

SQL

```
INSERT INTO Order_Details_Archive
   SELECT OrderNumber, ProductNumber,
       QuotedPrice, QuantityOrdered
FROM Order_Details
WHERE Order_Details.OrderNumber IN
   (SELECT OrderNumber
    FROM Orders
    WHERE Orders.OrderDate < '2018-01-01')
```

CH16_Archive_2017_Order_Details 在 Order_Details_Archive 表中插入的行（添加了 2499 行）

OrderNumber	ProductNumber	QuotedPrice	QuantityOrdered
1	1	$1,200.00	2
1	6	$635.00	3
1	11	$1,650.00	4
1	16	$28.00	1
1	21	$55.00	3
1	26	$121.25	5
1	40	$174.60	6
2	27	$24.00	4
2	40	$180.00	4
3	1	$1,164.00	5
<< 其他行 >>			

❖ **说明**　这两个查询都没有按我的建议做，即总是包含列名列表。我这样编写这两个查询旨在告诉你，在哪些情况下列名列表并非绝对必不可少的。

- **Entertainment Agency 数据库**

“新建一条表示顾客 Kendra Hernandez 的记录，其街道地址、城市、州、邮政编码和电话号码分别为 457 211th St NE、Bothell、WA、98200 和 555-3945。”

转换/整理　Insert into ~~the~~ customers ~~table in the columns~~ (customer first name, customer last name, customer street address, customer city, customer state, customer ZIP Code, ~~and~~ customer phone number) ~~the~~ values (‘Kendra’, ‘Hernandez’, ‘457 211th St NE’, ‘Bothell’, ‘WA’, ‘98200’, and ‘555-3945’)（在顾客表的名、姓、街道地址、城市、州、邮政编码和电话号码列中分别插入值 Kendra、Hernandez、457 211th St NE、Bothell、WA、98200 和 555-3945）

SQL
```
INSERT INTO Customers
    (CustFirstName, CustLastName,
     CustStreetAddress, CustCity, CustState,
     CustZipCode, CustPhoneNumber)
VALUES ('Kendra', 'Hernandez',
    '457 211th St NE', 'Bothell', 'WA',
    '98200', '555-3945')
```

CH16_Add_Customer 在 Customers 表中插入的行（添加了 1 行）

CustFirst Name	CustLast Name	CustStreet Address	Cust City	Cust State	Cust ZipCode	CustPhone Number
Kendra	Hernandez	457 211th St NE	Bothell	WA	98200	555-3945

“新增一个演出合约，顾客为 Matt Berg，演出组合为 Jazz Persuasion，演出时间为 2018 年 8 月 15 日和 16 日的晚上 7 点到 11 点，经纪人为 Karen Smith。”

❖ **说明**　如果你查看 Engagements 表，将发现需要从 Customers 表获取 Matt Berg 的顾客 ID，从 Entertainers 表获取 Jazz Persuasion 的演出组合 ID，从 Agents 表获取 Karen Smith 的经纪人 ID。你可使用 SELECT 表达式来获取这些值。

请在 FROM 子句中指定这 3 个表，但不指定连接条件。另外，别忘了根据 Entertainers 表中的每日费用以及 15%的佣金计算合约价格。这种做法之所以可行，是因为只有一个姓名为 Matt Berg 的顾客，只有一个姓名为 Karen Smith 的经纪人，只有一个艺名为 Jazz Persuasion 的演唱组合。如果有多个姓名为上述姓名的顾客或经纪人，将在 Engagements 表中插入多行。

转换/整理　Insert into ~~the~~ engagements ~~table into the~~ (customer ID, entertainer ID, agent ID, start date, end date, start time, end time, ~~and~~ contract price) ~~columns the~~ selection ~~of~~ customer ID, entertainer ID, agent ID, ~~and the values August 15, 2018~~ ‘2018-08-15’, ~~August 16, 2018~~ ‘2018-08-16’, ‘~~07:00:00 p.m.~~’ ‘19:00:00’, ‘~~11:00:00 p.m.~~’ ‘23:00:00’, ~~and the~~ (enter~~tainer~~ price per day ~~times~~ * 2 ~~times~~ * 1.15) from ~~the~~ customers, entertainers, ~~and~~ agents ~~tables~~ where ~~the~~ cust~~omer~~ first name ~~is~~ = ‘Matt’ and ~~the~~ cust~~omer~~ last name ~~is~~ = ‘Berg’ and ~~the~~ ent~~ertainer~~ stage name ~~is~~ = ‘Jazz Persuasion’ and ~~the~~ agen~~t~~ first name ~~is~~ = ‘Karen’ and ~~the~~ agen~~t~~ last name ~~is~~ = ‘Smith’（在顾客表、演唱组合表和经纪人表中，从顾客名为 Matt、顾客名为 Berg、演唱组合艺名为 Jazz Persuasion、经纪人名为 Karen、经纪人姓为 Smith 的行中选择顾客 ID、演唱组合 ID、经纪人 ID 以及值 2018-08-15、2018-08-16、19:00:00、23:00:00、演唱组合每日价格与 2 和 1.15 的乘积，将它们分别插入演唱合约表中的顾客 ID、演唱组合 ID、经纪人 ID、开始日期、结束日期、开始时间、结束时间和合约价格列中）

SQL
```
INSERT INTO Engagements
    (CustomerID, EntertainerID, AgentID,
     StartDate, EndDate,
     StartTime, StopTime,
     ContractPrice)
   SELECT Customers.CustomerID,
       Entertainers.EntertainerID, Agents.AgentID,
       '2018-08-15', '2018-08-16',
       '19:00:00', '23:00:00',
       ROUND(EntPricePerDay * 2 * 1.15, 0)
```

```
            FROM Customers, Entertainers, Agents
            WHERE (Customers.CustFirstName = 'Matt')
              AND (Customers.CustLastName = 'Berg')
              AND (Entertainers.EntStageName =
                      'Jazz Persuasion')
              AND (Agents.AgtFirstName = 'Karen')
              AND (Agents.AgtLastName = 'Smith')
```

❖ **说明** 你可能注意到了，我在 FROM 子句中指定了 3 个表，但没有连接它们。当你这样做时，将得到第一个表中各行、第二个表中各行和第三个表中各行的所有组合，这被称为**笛卡儿积**。在这里，这是可行的，因为这将筛选出特定的顾客、演唱组合和经纪人组合，即从每个表中取出一行，进而导致 SELECT 语句返回一行。第 20 章将更详细地介绍这种使用表的方式。

CH16_Add_Engagement 在 Engagements 表中插入的行（添加了 1 行）

Customer ID	Entertainer ID	Agent ID	Start Date	End Date	Start Time	Stop Time	Contract Price
10006	1005	4	2018-08-15	2018-08-16	19:00:00	23:00:00	$288.00

- **School Scheduling 数据库**

“新建一个名为 Italian 的科目类别，其科目编码为 ITA，属于人类学（Humanities）系。”

❖ **说明** 需要获悉人类学（Humanities）系的系 ID，因此必须使用一个检索 Departments 表的 SELECT 表达式。

转换/整理 Insert into ~~the~~ categories ~~table the~~ selection~~ of~~ ‘ITA’ as ~~the~~ category ID, ‘Italian’ as ~~the~~ category description, ~~and~~ department ID from ~~the~~ departments ~~table~~ where department name ~~is~~ = ‘Humanities’［从系表中选择 ITA（作为类别 ID）、Italian（作为类别描述）、系名 Humanities 对应的系 ID，将它们插入到类别表中］

SQL
```
INSERT INTO Categories
   SELECT 'ITA' AS CategoryID,
       'Italian' AS CategoryDescription,
      Departments.DepartmentID
   FROM Departments
   WHERE Departments.DeptName = 'Humanities'
```

CH16_Add_Category 在 Categories 表中插入的行（添加了 1 行）

CategoryID	CategoryDescription	DepartmentID
ITA	Italian	3

“新增一门课程，其科目 ID 为 4（Intermediate Accounting），教室为 3315，学分 5 个，从 2018 年 1 月 16 日开始周二和周四上课，开始时间为下午 3 点，时长 80 分钟。”

❖ **说明** 可假定“周一上课”“周二上课”等列的默认值为 0（假），因此只需将“周二上课”和“周四上课”的值设置为 1（真）。

转换/整理 Insert into ~~the~~ classes ~~table into the columns~~ (subject ID, classroom ID, credits, start date, start time, duration, Tuesday schedule, ~~and~~ Thursday schedule) ~~the~~ values (4, 3315, 5, ~~January 16, 2018~~ ‘2018-01-16’, ~~3 PM~~ ‘15:00:00’, 80, 1, ~~and~~ 1)（在课程表的科目 ID、教室 ID、学分、开始日期、开始时间、时长、周二上课和周四上课列中分别插入值 4、3315、5、2018-01-16、15:00:00、80、1 和 1）

SQL
```
INSERT INTO Classes
      (SubjectID, ClassRoomID, Credits,
      StartDate, StartTime, Duration,
      TuesdaySchedule, ThursdaySchedule)
   VALUES (4, 3315, 5, '2018-01-16', '15:00:00',
      80, 1, 1)
```

CH16_Add_New_Accounting_Class 在 Classes 表中插入的行（添加了 1 行）

Subject ID	ClassRoom ID	Credits	Start Date	Start Time	Duration	Tuesday Schedule	Thursday Schedule
4	3315	5	2018-01-16	15:00:00	80	1	1

- **Bowling League 数据库**

"通过复制表示 Neil Patterson 的记录，新建一条表示 Matthew Patterson 的投球手记录。"

❖ **说明**　务必将总得分、参赛的局数、当前平均得分和当前让分数设置为零。

转换/整理

Insert into ~~the~~ bowlers ~~table into the columns~~ (bowler last name, bowler first name, bowler address, bowler city, bowler state, bowler zip, bowler phone number, team ID, bowler total pins, bowler games bowled, bowler current average, ~~and~~ bowler current handicap) ~~the~~ select~~ion of~~ bowler last name, ~~the value~~ 'Matthew', bowler address, bowler city, bowler state, bowler zip, bowler phone number, team ID, ~~and the values~~ 0, 0, 0, ~~and~~ 0 from ~~the~~ bowlers ~~table~~ where ~~the~~ bowler last name ~~is~~ = 'Patterson' and ~~the~~ bowler first name ~~is~~ = 'Neil'（在投球手表中，从投球手名为 Patterson 且投球手姓为 Neil 的行中选择姓、值 Matthew、地址、城市、州、邮政编码、电话号码、球队 ID 以及值 0、0、0、0，并将它们分别插入到投球手表的姓、名、地址、城市、州、邮政编码、电话号码、球队 ID、总得分、参赛局数、当前平均得分、当前让分数等列中）

SQL

```
INSERT INTO Bowlers
    (BowlerLastName, BowlerFirstName,
     BowlerAddress, BowlerCity,
     BowlerState, BowlerZip,
     BowlerPhoneNumber, TeamID,
     BowlerTotalPins, BowlerGamesBowled,
     BowlerCurrentAverage, BowlerCurrentHcp)
  SELECT Bowlers.BowlerLastName, 'Matthew',
       Bowlers.BowlerAddress, Bowlers.BowlerCity,
       Bowlers.BowlerState, Bowlers.BowlerZip,
       Bowlers.BowlerPhoneNumber, Bowlers.TeamID,
       0, 0,
       0, 0
   FROM Bowlers
   WHERE (Bowlers.BowlerLastName = 'Patterson')
       AND (Bowlers.BowlerFirstName = 'Neil')
```

CH16_Add_Bowler 在 Bowlers 表中插入的行（添加了 1 行）

Bowler LastName	Bowler FirstName	Bowler Address	Bowler City	Bowler State	Bowler Zip
Patterson	Matthew	16 Maple	Auburn	WA	98002
Bowler PhoneNumber	**TeamID**	**Bowler TotalPins**	**Bowler GamesBowled**	**BowlerCurrent Average**	**Bowler CurrentHcp**
(206) 555-3487	2	0	0	0	0

"新增一个名为 Aardvarks 的球队，该球队没有指定队长。"

转换/整理

Insert into ~~the~~ teams ~~table into the columns~~ (team name, ~~and~~ captain ID) ~~the~~ values ('Aardvarks', ~~and~~ Null)（在球队表的球队名称和队长 ID 列中插入值 Aardvarks 和 Null）

SQL

```
INSERT INTO Teams
    (TeamName, CaptainID)
VALUES ('Aardvarks', NULL)
```

CH16_Add_Team 在 Teams 表中插入的行（添加了 1 行）

TeamName	CaptainID
Aardvarks	NULL

16.5 小结

本章首先简要地讨论了用于在表中添加数据的 INSERT 语句。我们介绍了 INSERT 语句的语法，并列举了一个使用值列表添加一行的示例。

接下来讨论了大多数数据库系统具备的一种功能，它让你能够生成下一个独一无二的值，用作新增行的主键值。为了支持这种功能，Microsoft SQL Server 提供了数据类型 Identity，Microsoft Access 提供了数据类型 AutoNumber，MySQL 提供了属性 AUTO_INCREMENT，而 Oracle 数据库系统提供了 Sequence 伪列。此外还介绍了如何在 VALUES 子句中使用子查询来获取现有的最大主键值，再加上 1。

然后探索了如何在 INSERT 语句中使用 SELECT 表达式复制一行或多行，回顾了 SELECT 表达式的语法，演示了如何从一个表中复制一行到另一个表中，通过一个将旧记录复制到归档表的示例探索了如何复制多行。

接下来演示了在需要从相关表中获取一个或多个值，以便将其插入到当前表中时，SELECT 表达式很有用。

最后，本章列举了一些 INSERT 查询编写示例。

下一节列出了多个你可独自解决的问题。

16.6 练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 顾客 Liz Keyser 想再次订购他在 2017 年 12 月 12 日订购的商品；下单日期为 2018 年 6 月 12 日，送货日期为 2018 年 6 月 15 日。

  提示：需要依次复制 Orders 和 Order_Details 表中的行。假定可这样生成新订单的订单号：找到 Liz Keyser 在 2017 年 12 月 12 日所下的订单，获取其 OrderID 列的值并加上 1000。如果你使用的是 Microsoft SQL Server，务必在向 Orders 表中插入数据前执行命令 SET IDENTITY_INSERT Orders ON。如果你使用的是 PostgreSQL，务必在插入后使用 SetVal 设置 Orders_OrderNumber_seq 的值。

  解决方案见 CH16_Copy_Dec12_Order_For_Keyser（添加了 1 行）和 CH16_Copy_Dec12_OrderDetails_For_Keyser（添加了 4 行）。

  > ❖ **说明**　不推荐通过加上一个固定的值来生成新的主键值，因为要这样做，必须先确定至少加上多少才能确保查询可行。实际上，要避免这样做，可使用数据库系统提供的函数来返回下一个可供你使用的合法值。然而，这种编程超出了本书的范围。这里让你加上一个数字旨在简化问题解决方案，但你必须知道这种做法并不是最佳的。

  (2) 新建一条表示顾客 Mary Baker 的记录，其街道地址、城市、州、邮政编码、区号和电话号码分别为 7834 W 32nd Ct、Bothell、WA、98011、425 和 555-9876。

  解决方案见 CH16_Add_Sales_Customer（添加了 1 行）。

  (3) 顾客 Angel Kennedy 想再次订购她在 2017 年 11 月订购的所有商品；下单日期为 2018 年 6 月 15 日，送货日期为 2018 年 6 月 18 日。

  提示：需要依次复制 Orders 和 Order_Details 表中的行。假定可这样生成新订单的订单号：找到 Angel Kennedy 在 2017 年 11 月所下的订单，获取其 OrderID 列的值并加上 1000。如果你使用的是 Microsoft SQL Server 或 PostgreSQL，请参阅前面的问题 1 中的提示。

  解决方案见 CH16_Copy_November_Orders_For_AKennedy（添加了 6 行）和 CH16_Copy_November_OrderDetails_For_AKennedy（添加了 34 行）。

- **Entertainment Agency 数据库**

  (1) 经纪人 Marianne Wier 想与演唱组合签订演出合约，因此需要新建一条顾客记录，方法是复制经纪人表中相关列中的信息。
  提示：只需将 Agents 表中的相关列复制到 Customers 表中。
  解决方案见 CH16_Copy_Agent_To_Customer（添加了 1 行）。

  (2) 新增音乐风格 New Age。
  解决方案见 CH16_Add_Style（添加了 1 行）。

  (3) 顾客 Doris Hartwig 想再次与一个演唱组合签约，她请该演唱组合在 2017 年 12 月 1 日演出过，但这次的演出日期为 2018 年 8 月 1 日。
  提示：使用一个 SELECT 表达式，它连接 Customers 和 Engagements 表，并以字面量值的方式提供新的演出日期。
  解决方案见 CH16_Duplicate_Engagement（添加了 1 行）。

- **School Scheduling 数据库**

  (1) Angel Kennedy 想注册成为学生。她丈夫 John 已注册，请使用 John 的信息新建一条表示 Angel 的学生记录。
  解决方案见 CH16_Add_Student（添加了 1 行）。

  (2) 教员 Tim Smith 想注册成为学生，请根据他的教员记录新建一条学生记录。
  解决方案见 CH16_Enroll_Staff（添加了 1 行）。

- **Bowling League 数据库**

  (1) 对于 2017 年举行的所有比赛场次，将其联赛 ID、场次、局次和投球手得分归档。
  提示：需要编写 4 个查询，分别将 Tournaments、Tourney_Matches、Match_Games 和 Bowler_Scores 表中的相关行归档。为遵守引用完整性规则，必须先复制 Tournaments，再复制 Tourney_Matches，然后复制 Match_Games，最后复制 Bowler_Scores。
  解决方案见 CH16_Archive_2017_Tournaments_1（添加了 14 行）、CH16_Archive_2017_Tournaments_2（添加了 57 行）、CH16_Archive_2017_Tournaments_3（添加了 168 行）和 CH16_Archive_2017_Tournaments_4（添加了 1344 行）。

  (2) 新建 2019 年的联赛和联赛场次，方法是复制 2017 年对应周的联赛和联赛场次。
  提示：假定可将 2017 年联赛的 TourneyID 列的值加上 25 来生成新的联赛 ID。需要复制 Tournaments 和 Tourney_Matches 表中的行。如果你使用的是 Microsoft SQL Server，务必在 Tournaments 表中插入数据前执行命令 SET IDENTITY_INSERT Tournaments ON；如果你使用的是 PostgreSQL，务必在插入后使用 SetVal 设置 Tournaments_TournamentID_seq 的值。
  另外请注意，不能简单地给日期加上 2 年或 730 天，因为这里的目标在同一周的同一天举办相应的联赛，而 52 周乘以每周 7 天为 364 天，而不是 365 天。还有，务必在两条语句中都筛选出正确的日期，为此需要在第二条语句中使用子查询。
  解决方案见 CH16_Copy_2017_Tournaments_1（添加了 14 行）和 CH16_Copy_2017_Tournaments_2（添加了 57 行）。

第 17 章

删除数据集

“最终我爱上了一行行的豆子，尽管它们的数量大大地超出了我的需要。”

——亨利·戴维·梭罗

本章涵盖如下主题：

- ❑ 何谓删除
- ❑ DELETE 语句
- ❑ DELETE 的用途
- ❑ 语句举例
- ❑ 小结
- ❑ 练习

至此，你知道了如何使用 UPDATE 语句修改数据，还学习了如何使用 INSERT 语句添加数据，但如何删除不需要的数据呢？为此，需要使用最简单但也最危险的 SQL 语句——DELETE。

17.1 何谓删除

通过前一章的学习你知道，在表中添加数据非常简单。可使用 VALUES 子句每次添加一行，也可使用 SELECT 表达式复制多行。但如果你错误地添加了一行，该怎么办呢？如何删除你已复制到归档表中的行呢？如何删除从未下单的顾客呢？如何删除已申请入学但没有注册任何课程的学生呢？如果要将表变成空的，如何删除其中所有的行呢？所有这些问题的答案都是“使用 DELETE 语句”。与其他 SQL 语句一样，DELETE 语句处理的也是行集。本章你将学到，最简单的 DELETE 语句删除指定表中所有的行，但在大多数情况下，你只想删除部分行。如果你猜测为此可使用 WHERE 子句，那么你完全正确。

❖ **说明** 所有示例语句和解决方案都可在相应示例数据库的“修改”版本——Sales Orders Modify、Entertainment Agency Modify、School Scheduling Modify 和 Bowling League Modify 中找到。

17.2 DELETE 语句

DELETE 语句只有 3 个关键字——DELETE、FROM 和 WHERE，其语法图如图 17-1 所示。

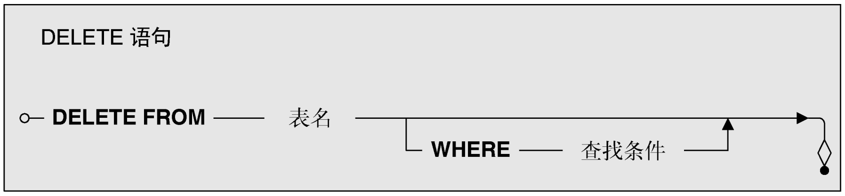


图 17-1　DELETE 语句的语法图

前面说过，DELETE 语句可能是最简单的 SQL 语句，这可不是开玩笑！但在你能够执行的语句中，它也是最危险的。如果没有包含 WHERE 子句，DELETE 语句将删除指定表中所有的行，这在测试新应用程序时很有用，它让你能够清空既有表中所有的行，但保留表结构。在你设计的应用程序中，还可能有工作表（临时表），你在其中加载数据以执行特定的任务。例如，经常需要使用 INSERT 语句将非常复杂的 SELECT 表达式返回的行复制到表中，再使用这个表来生成多个静态报告。在这种情况下，不包含 WHERE 子句的 DELETE 就可派上用场——在生成新报告前使用它来清空旧行。

❖ **说明**　SQL 标准指出，表名也可以是查询（视图）名，但查询名指定的表必须是可更新的。很多数据库系统都支持从视图中删除行；在可更新的视图必须满足什么条件方面，每个数据库系统有不同的规定。在大多数情况下，如果使用了关键字 DISTINCT 或某个输出列是由表达式或聚合函数生成的，视图就是不可更新的。

有些数据库系统还支持在定义视图（在 SQL 标准中被称为派生表）时使用关键字 JOIN 和 ON，而不是表名。在支持使用派生表的系统中，必须紧跟在关键字 FROM 后面，以“表名.*”的形式将一个参与连接的表指定为删除目标。有关这方面的细节，请参阅你使用的数据库系统的文档。在本章的每条 DELETE 语句中，都只将单个表作为目标。

17.2.1　删除所有行

删除所有的行实在是太容易了。下面编写一条 DELETE 语句，对处理示例数据库 Bowling League 中的 Bowlers 表进行处理。

❖ **说明**　贯穿本章都将采用第 4 章介绍的“请求/转换/整理/SQL”流程。

“删除所有投球手。”

转换	Delete all rows from the bowlers table（删除投球手表中所有的行）
整理	Delete ~~all rows~~ from ~~the~~ bowlers ~~table~~
SQL	`DELETE` `FROM Bowlers`

如果你在示例数据库中执行这条语句，它真的会删除所有的行吗？答案是否定的，因为我在 Bowlers 和 Bowler_Scores 之间定义了一个约束条件（第 2 章讨论的引用完整性规则）：对于特定的投球手，只要 Bowler_Scores 表中有与之相关的行，数据库系统就不会允许你将其从 Bowlers 表中删除。

在示例数据库 Bowling League 的修改版中，有两名投球手没有任何得分，因此使用这条简单的 DELETE 语句应该能够删除他们。即便你本不想删除任何行，这两行也将永远消失，准确地说，是可能会永久消失。首先，很多数据库系统都维护着表修改日志，有时可从该系统日志恢复丢失的数据。另外，第 15 章讨论事务时说过，如果你启动了事务（或者系统替你启动了），就可在遇到错误时回滚所有未提交的修改。

你可能还记得我说过，Microsoft Office Access 是一个这样的数据库系统，即每当你从用户程序界面执行查询时，它都会替你自动启动事务。如果你在 Microsoft Access 中运行这个查询，它将告诉你将删除多少行。此时，如果你发现数据库系统将删除表中的所有行，可撤销删除操作。如果你让系统继续做下去，将出现如图 17-2 所示的错误对话框。

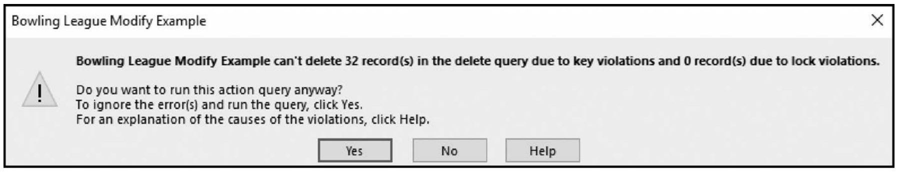


图 17-2　有些数据库系统会在你执行将导致错误的 DELETE 语句时发出警告

从中可知，由于“键违规”，这个表中 34 条记录中的 32 条不会被删除。这相当于委婉地告诉你：“嗨，傻瓜，Bowler_Scores 表中还有与你要删除的投球手相关的行呢。”如果你此时单击 No，数据库系统将替你执行回滚（ROLLBACK），因此不会删除任何行；如果你单击 Yes，数据库系统将执行提交（COMMIT），永久性地删除那两个没有任何得分的投球手。

17.4 节将介绍两种将没有参加任何比赛的投球手删除的安全方法。

17.2.2　删除某些行

大多数情况下，你想限制要删除的行。为此，可添加一个 WHERE 子句，将要删除的行筛选出来。与在 SELECT 和 UPDATE 语句中一样，这个 WHERE 子句可以非常简单，也可以非常复杂。

1. 使用简单的 WHERE 子句

我们从简单的着手。假设你要从数据库 Sales Orders 中删除所有总价为零的订单，这个请求可能类似于下面这样。

“删除总价为零的订单。”

转换	Delete from the orders table where the order total is zero（从订单表中删除总价为零的订单）
整理	Delete from ~~the~~ orders ~~table~~ where ~~the~~ order total ~~is~~ = ~~zero~~ 0
SQL	`DELETE FROM Orders` `WHERE OrderTotal = 0`

其中的 WHERE 子句使用一个简单的比较谓词来找出订单总价为零的行。如果你在示例数据库中执行这个查询，将发现它删除了 11 行。我保存了这个查询，名为 CH17_Delete_Zero_OrdersA。

2. 安全第一：确保将删除正确的行

即便是简单的 DELETE 查询，也强烈建议你核实它将删除正确的行。如何核实呢？前面说过，在大多数情况下，你会添加 WHERE 子句来选择部分行以便将其删除。为何不先编写一个 SELECT 查询来返回你要删除的行呢？

“列出 Orders 表中总价为零的订单的所有列。”

转换	Select all columns from the orders table where the order total is zero（在订单表中，从总价为零的行中选择所有的列）
整理	Select ~~all columns~~ * from ~~the~~ orders ~~table~~ where ~~the~~ order total ~~is~~ = ~~zero~~ 0
SQL	`SELECT * FROM Orders` `WHERE OrderTotal = 0`

如果在示例数据库 Sales Orders 中运行这个查询，结果应类似于图 17-3。

OrderNumb	OrderDate	ShipDate	CustomerID	EmployeeID	OrderTotal
198	2017-10-08	2017-10-10	1002	703	$0.00
216	2017-10-12	2017-10-12	1016	707	$0.00
305	2017-11-01	2017-11-05	1013	708	$0.00
361	2017-11-12	2017-11-13	1016	706	$0.00
484	2017-12-09	2017-12-10	1021	707	$0.00
523	2017-12-15	2017-12-17	1003	704	$0.00
629	2018-01-08	2018-01-12	1014	704	$0.00
632	2018-01-08	2018-01-12	1001	706	$0.00
689	2018-01-15	2018-01-16	1015	705	$0.00
753	2018-01-28	2018-01-30	1013	701	$0.00
816	2018-02-09	2018-02-12	1011	701	$0.00

图 17-3　查看你要删除的行

请注意，我使用了简写字符*指出要查看所有的列。如果结果集中包含你要删除的所有行，就可将 SELECT 语句转换为正确的 DELETE 语句，为此只需将 SELECT *替换为 DELETE。图 17-4 说明了如何将这条 SELECT 语句转换为正确的 DELETE 语句。

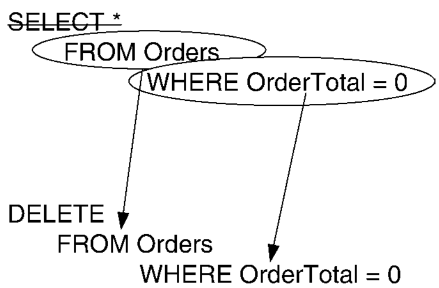


图 17-4　将用于验证的 SELECT 查询转换为 DELETE 语句

这种转换是如此地简单，以至于如果不先编写 SELECT 语句来确保你将删除正确的行，都显得有点傻了。别忘了，除非在事务中执行 DELETE 语句，否则执行它后，被删除的行将“一去不复返”。

3. 使用子查询

前一节编写的查询删除总价为零的所有订单，看起来足够简单。但别忘了，OrderTotal 列是一个计算列（第 15 章演示了如何使用 UPDATE 查询计算并设置总价）。如果添加、修改或删除一个或多个订单详情行后，用户或应用程序没有运行 UPDATE 查询呢？前述简单的查询将试图删除在 Order_Details 表中有相关行的订单，还可能遗漏相关订单详情行都已删除但总价未更新的订单。

为了确保你要删除的订单没有相关的订单详情行，一种更安全的方式是，使用一个子查询从 Order_ Details 表中获取相关行。这样请求可能类似于下面这样：

“删除所有不包含任何商品的订单。”

转换	Delete the rows from the orders table where the order number is not in the selection of the order number from the order details table（从订单详情表中选择订单号，生成一个列表；从订单表中删除订单号不在上述列表中的行）
整理	Delete ~~the rows~~ from ~~the~~ orders ~~table~~ where ~~the~~ order number ~~is~~ not in ~~the~~ (selec~~tion of the~~ order number from ~~the~~ order details) ~~table~~

SQL	`DELETE FROM Orders`
	`WHERE OrderNumber NOT IN`
	`  (SELECT OrderNumber`
	`   FROM Order_Details)`

与将订单总价同零进行简单比较相比，这要复杂些，但可确保只删除在 Order_Details 表中没有匹配行的订单。这个查询保存到了相应的示例数据库中，名为 CH17_Delete_Zero_OrdersB。这个更复杂的查询可能找到并删除一些总价不为零的订单，出现这种情况的原因是这些订单包含的最后一件商品被删除时，没有更新总价。

给 DELETE 查询编写 WHERE 子句时，你可能经常用到 IN、NOT IN、EXISTS 或 NOT EXISTS（如果需要复习这些谓词，请参阅第 11 章）。再来看一个例子，它也需要使用复杂的 WHERE 子句来筛选出要删除的行。

“删除 2018 年 1 月 1 日之前的所有订单及其订单详情（这些信息已复制到归档表中）。”

第 16 章介绍了如何使用 INSERT 将旧的行集复制到一个或多个归档表中。归档这些行后，如果将它们删除，通常可提高应用程序主体部分的处理效率。这个请求表明，需要从两个表中删除行，因此我们将其分为两个请求。需要先从 Order_Details 表中删除行，因为对于 Orders 表中的行，如果在 Order_Details 表中有与之匹配的行，定义的引用完整性规则就不会让你删除它。

“删除 2018 年 1 月 1 日之前的所有订单的订单详情（这些信息已复制到归档表中）。”

你看到潜在的风险了吗？对于这个问题，一种解决方式是直接将 2018 年 1 月 1 日之前的所有订单的订单详情行删除。

转换	Delete rows from the order details table where the order number is in the selection of the order number from the orders table where the order date is earlier than January 1, 2018（在订单表中，从下单日期在早于 2018 年 1 月 1 日的行中选择订单号，生成一个列表；从订单详情表中删除订单号在上述列表中的行）
整理	Delete ~~rows~~ from ~~the~~ order details ~~table~~ where ~~the~~ order number ~~is~~ in ~~the~~ (selec~~tion of the~~ order number from ~~the~~ orders ~~table~~ where ~~the~~ order date ~~is earlier than~~ < ~~January 1, 2018~~ ‘2018-01-01’)

SQL

```
DELETE FROM Order_Details
WHERE OrderNumber IN
   (SELECT OrderNumber
    FROM Orders
    WHERE OrderDate < '2018-01-01')
```

我保存了这个查询，并将其命名为 CH17_Delete_Archived_Order_Details_Unsafe。如果承诺运行 INSERT 查询以将这些行归档的人并没有付诸行动呢？如果你运行这个查询，将删除 2018 年 1 月 1 日之前的所有订单的订单详情，而不管它们是否在归档表中。一种更安全的方式是，只删除那些确实在归档表中的行。我们来尝试这样做。

转换	Delete rows from the order details table where the order number is in the selection of order number from the order details archive table（从订单详情归档表中选择订单号，生成一个列表；从订单详情表中删除订单号在上述列表中的行）
整理	Delete ~~rows~~ from ~~the~~ order details ~~table~~ where ~~the~~ order number ~~is~~ in ~~the~~ (selec~~tion of~~ order number from ~~the~~ order details archive) ~~table~~

SQL

```
DELETE FROM Order_Details
WHERE OrderNumber IN
   (SELECT OrderNumber
    FROM Order_Details_Archive)
```

我保存了这个查询，并将其命名为 CH17_Delete_Archived_Order_Details_OK。请注意，这个查询根本不关心下单日期，但安全得多，因为它只删除主表中这样的行，即在归档表中有与之匹配的订单行。如果你要确保删除的是与 2018 年 1 月 1 日之前的订单相关的行，可在子查询中使用布尔运算符合并两个 IN 谓词。

17.3　DELETE 的用途

至此，你应该对如何从表中删除一行或多行（所有行或使用 WHERE 子句筛选出的行）有了深入的认识。为了展示 DELETE 语句的用途有多广泛，最佳的方式是列举一些使用它可解决的问题，并提供一些使用示例（参见 17.4 节）。下面是使用 DELETE 可解决的问题类型中的很小一部分。

“删除从未被订购过的商品。”
“删除从未签约的演唱组合。”
“删除从未参加过比赛的投球手。”
“删除没有注册任何课程的学生。”
“删除不包含任何商品的类别。”
“删除从未邀请演唱组合去演出的顾客。”
“删除没有任何投球手的球队。”
“删除从未有学生注册的课程。”
“删除没下过订单的顾客。”
“删除没有任何演唱组合演奏的音乐风格。”
“删除没有举行的保龄球比赛场次。”
“删除不包含任何课程的科目。”
“删除已复制到归档表中的演出合约。”
“删除已复制到归档表中的联赛数据。”
“删除不提供任何商品的供应商。”
“删除不属于任何演唱组合的成员。”
“删除未售出任何商品的员工。”

17.4 语句举例

知道编写 DELETE 查询的机制后，来看一系列示例，它们都要求从表中删除一行或多行，且都来自 4 个示例数据库。

> ❖ **警告** 在修改版示例数据库中，所有的示例查询都会修改数据，因此有些查询仅在第一次运行时的效果与预期相同。例如，如果你运行一个 DELETE 查询来删除订单，而它使用了 WHERE 子句来查找要删除的行，那么再次运行时将失败，因为要删除的行在第一次运行查询时被删除了。如果你要多次练习解决问题，可使用示例脚本或备份恢复数据库。

在每个示例中，都列出了示例 DELETE 语句删除的数据，指出了它删除了多少行。在删除的行数前面，指出了在本书配套网站提供的示例数据库中，保存查询时使用的名称。另外，对于每个 DELETE 查询，我还编写了配套的 SELECT 查询（在 MySQL 和 Microsoft SQL Server 中存储为视图），你可使用它们来获悉将删除哪些数据。给这些配套查询命名时，在 DELETE 查询的名称后面添加了_Query。我将每个查询都存储到了相应的示例数据库中，名称以 CH17 打头。示例数据库可在本书配套网站中找到。你可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明** 别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录 B。为了简化查询编写过程，我在下面的示例中将转换和整理合二为一了。这些示例都假定你深入研究并掌握了本书前面介绍的概念。

- **Sales Orders 数据库**

“删除没下过订单的顾客。”

转换/整理	Delete ~~rows~~ from ~~the~~ customers ~~table~~ where ~~the~~ customer ID ~~is~~ not in ~~the~~ (select~~ion of the~~ customer ID from ~~the~~ orders) ~~table~~ （从订单表中选择顾客 ID，生成一个列表；从顾客表中删除顾客 ID 不在上述列表中的行）

SQL

```
DELETE
FROM Customers
WHERE CustomerID NOT IN
  (SELECT CustomerID
   FROM Orders)
```

CH17_Delete_Customers_Never_Ordered 从 Customers 表中删除的行（删除了 1 行）

Customer ID	Cust First Name	Cust Last Name	Cust Street Address	Cust City	Cust State	Cust Zip Code	Cust Area Code	Cust Area Code
1028	Jeffrey	Tirekicker	15622 NE 42nd Ct	Redmond	WA	98052	425	555-9999

❖ **说明**　如果你执行了前一章的查询 CH16_Add_Customer，将删除两行，其中第 2 行表示的是顾客 Kendra Hernandez。

“删除不提供任何商品的供应商。”

转换/整理　Delete ~~rows~~ from ~~the~~ vendors ~~table~~ where ~~the~~ vendor ID ~~is~~ not in ~~the~~ (selec~~tion of~~ vendor ID from ~~the~~ product vendors) ~~table~~（从商品供应商表中选择供应商 ID，生成一个列表；从供应商表中删除供应商 ID 不在上述列表中的行）

SQL

```
DELETE
FROM Vendors
WHERE VendorID NOT IN
    (SELECT VendorID
     FROM Product_Vendors)
```

CH17_Delete_Vendors_No_Products 从 Vendors 表中删除的行（删除了 1 行）

Vendor ID	VendName	VendName VendStreet Address	Vend City	Vend State	Vend ZipCode	<<其他列>>
11	Astro Paper-b Products	5639N. Riverside	Chbicago	IL	60637	…

❖ **说明**　如果你执行了前一章的查询 CH16_Add_Vendor，将删除两行，其中第 2 行表示的是供应商 Hot Dog Bikes。

- **Entertainment Agency 数据库**

“删除从未签约的演唱组合。”

❖ **说明**　要从 Entertainer 表中删除任何行，必须先从 Entertainer_Members 和 Entertainer_Styles 表中删除所有与之相关的行。

第 1 个转换/整理　Delete ~~rows~~ from ~~the~~ entertainer members ~~table~~ where ~~the~~ entertainer ID ~~is~~ not in ~~the~~ (selec~~tion of~~ entertainer ID from ~~the~~ engagements) ~~table~~（从演出合约表中选择演唱组合 ID，生成一个列表；从演唱组合成员表中删除演唱组合 ID 不在上述列表中的行）

SQL

```
DELETE
FROM Entertainer_Members
WHERE EntertainerID NOT IN
    (SELECT EntertainerID
     FROM Engagements)
```

CH17_Delete_Entertainers_Not_Booked1 从 Entertainer_Members 表中删除的行（删除了 1 行）

EntertainerID	MemberID	Status
1009	121	2

第 2 个转换/整理：Delete ~~rows~~ from ~~the~~ entertainer styles ~~table~~ where ~~the~~ entertainer ID ~~is~~ not in ~~the~~ (selecti~~on of~~ entertainer ID from ~~the~~ engagements) ~~table~~（从演出合约表中选择演唱组合 ID，生成一个列表；从演唱组合风格表中删除演唱组合 ID 不在上述列表中的行）

SQL

```
DELETE
FROM Entertainer_Styles
WHERE EntertainerID NOT IN
   (SELECT EntertainerID
    FROM Engagements)
```

CH17_Delete_Entertainers_Not_Booked2 从 Entertainer_Styles 表中删除的行（删除了 3 行）

EntertainerID	MemberID
1009	7
1009	14
1009	21

第 3 个转换/整理：Delete ~~rows~~ from ~~the~~ entertainers ~~table~~ where ~~the~~ entertainer ID ~~is~~ not in ~~the~~ (selecti~~on of~~ entertainer ID from ~~the~~ engagements) ~~table~~（从演出合约表中选择演唱组合 ID，生成一个列表；从演唱组合表中删除演唱组合 ID 不在上述列表中的行）

SQL

```
DELETE
FROM Entertainers
WHERE EntertainerID NOT IN
   (SELECT EntertainerID
    FROM Engagements)
```

CH17_Delete_Entertainers_Not_Booked3 从 Entertainers 表中删除的行（删除了 1 行）

Entertainer ID	EntStage Name	Ent SSN	EntStreet Address	Ent City	Ent State	Ent ZipCode	<<其他列>>
1009	Katherine Ehrlich	888-61-1103	777 Fenexet Blvd	Woodinville	WA	98072	…

“删除已复制到归档表中的演出合约。”

转换/整理：Delete ~~rows~~ from ~~the~~ engagements ~~table~~ where ~~the~~ engagement ID ~~is~~ in ~~the~~ (selecti~~on of~~ engagement ID from ~~the~~ engagements archive) ~~table~~（从演出合约归档表中选择演出合约 ID，生成一个列表；从演出合约表中删除演出合约 ID 在上述列表中的行）

SQL

```
DELETE
FROM Engagements
WHERE EngagementID IN
   (SELECT EngagementID
    FROM Engagements_Archive)
```

❖ **说明**　要找出要删除的行，必须先运行查询 CH16_Archive_Engagements 将数据复制到归档表中。在示例数据库中，归档表最初是空的。

CH17_Remove_Archived_Engagements 从 Engagements 表中删除的行
（如果执行了查询 CH16_Archive_Engagements，将删除 56 行）

Engagement Number	Start Date	End Date	Start Time	Stop Time	Contract Price	Customer ID	Agent ID	Entertainer ID
4	2017-09-12	2017-09-18	20:00	0:00	$470.00	10007	3	1004
5	2017-09-12	2017-09-15	16:00	19:00	$1,130.00	10006	5	1003
6	2017-09-11	2017-09-15	15:00	21:00	$2,300.00	10014	7	1008

- **School Scheduling 数据库**

“删除从未有学生注册的课程。”

❖ **说明** 需要先删除 Faculty_Classes 表中的相关行，再删除 Classes 表中满足条件的行，因为这个数据库有一个引用完整性规则，不允许你删除 Classes 表中这样的行，即 Faculty_Classes 表中有与之匹配的行。

第 1 个转换/整理　Delete from ~~the~~ faculty classes ~~table~~ where ~~the~~ class ID ~~is~~ not in ~~the~~ (select~~ion of~~ class ID from ~~the~~ student schedules) ~~table~~（从学生选课表中选择课程 ID，生成一个列表；从教员课程表中删除课程 ID 不在上述列表中的行）

SQL

```
DELETE
FROM Faculty_Classes
WHERE ClassID NOT IN
   (SELECT ClassID
    FROM Student_Classes)
```

CH17_Delete_Classes_No_Students_1 从 Faculty_Classes 表中删除的行（删除了 113 行）

ClassID	StaffID
1002	98036
1002	98036
1004	98019
1006	98045
1012	98030
1031	98005
1183	98005
1184	98011
1196	98028
1560	98028
<< 其他行 >>	

第 2 个转换/整理　Delete from ~~the~~ classes ~~table~~ where ~~the~~ class ID ~~is~~ not in ~~the~~ (select~~ion of~~ class ID from ~~the~~ student schedules) ~~table~~（从学生选课表中选择课程 ID，生成一个列表；从课程表中删除课程 ID 不在上述列表中的行）

SQL

```
DELETE
FROM Classes
WHERE ClassID NOT IN
   (SELECT ClassID
    FROM Student_Schedules)
```

CH17_Delete_Classes_No_Students_2 从 Classes 表中删除的行（删除了 115 行）

Class ID	Subject ID	Classroom ID	Credits	Start Date	Start Time	Duration	Monday Schedule	<<其他列>>
1002	12	1619	4	2017-09-11	15:30	110	Yes	…
1004	13	1627	4	2017-09-11	8:00	50	Yes	…
1006	13	1627	4	2017-09-11	9:00	110	Yes	…
1012	14	1627	4	2017-10-11	13:00	110	No	…
1031	16	1231	5	2017-09-11	14:00	50	Yes	…
1183	38	3415	5	2017-09-11	13:00	50	Yes	…
1184	38	3415	5	2017-09-11	14:00	50	Yes	…
1196	39	3415	5	2017-09-11	15:00	50	Yes	…
<< 其他行 >>								

- **Bowling League 数据库**

“删除从未参加过比赛的投球手。”

> ❖ **说明**　要解决这个问题，可删除比赛局数为零的投球手，也可删除在 Bowler_Scores 表中没有匹配行的投球手。第二种方法更安全，因为它不依赖于计算得到的值——比赛局数。但这里将演示这两种方式。

第 1 个转换/整理　Delete ~~rows~~ from ~~the~~ bowlers ~~table~~ where ~~the~~ bowler games bowled ~~is~~ = ~~zero~~ 0（从投球手表中删除比赛局数为零的行）

SQL

```
DELETE
FROM Bowlers
WHERE BowlerGamesBowled = 0
```

CH17_Delete_Bowlers_No_Games 从 Bowlers 表中删除的行（删除了 2 行）

Bowler ID	Bowler LastName	Bowler FirstName	<<其他列>>	Bowler Games Bowled	Bowler Current Average	Bowler Current Hcp
33	Patterson	Kerry	…	0	0	0
34	Patterson	Maria	…	0	0	0

第 2 个转换/整理　Delete ~~rows~~ from ~~the~~ bowlers ~~table~~ where ~~the~~ bowler ID ~~is~~ not in ~~the~~ (selecti~~on of~~ bowler ID from ~~the~~ bowler scores) ~~table~~（从投球手得分表中选择投球手 ID，生成一个列表；从投球手表中删除投球手 ID 不在上述列表中的行）

SQL

```
DELETE
FROM Bowlers
WHERE BowlerID NOT IN
   (SELECT BowlerID
    FROM Bowler_Scores)
```

CH17_Delete_Bowlers_No_Games_Safe 从 Bowlers 表中删除的行（删除了 2 行）

Bowler ID	Bowler LastName	Bowler FirstName	<<其他列>>	Bowler Games Bowled	Bowler Current Average	Bowler Current Hcp
33	Patterson	Kerry	…	0	0	0
34	Patterson	Maria	…	0	0	0

> ❖ **说明**　如果你执行了前一章的查询 CH16_Add_Bowler，这里的两个查询都将删除 3 行，其中第 3 行表示的是 Matthew Patterson。

“删除没有任何投球手的球队。”

转换/整理　Delete from ~~the~~ teams ~~table~~ where ~~the~~ team ID ~~is~~ not in ~~the~~ (selecti~~on of~~ team ID from ~~the~~ bowlers) ~~table~~（从投球手表中选择球队 ID，生成一个球队列表；从球队表中选择球队 ID 不在上述列表中的行）

SQL

```
DELETE
FROM Teams
WHERE TeamID NOT IN
   (SELECT TeamID
    FROM Bowlers)
```

CH17_Delete_Teams_No_Bowlers 从 Bowlers 表中删除的行（删除了 2 行）

TeamID	TeamName	CaptainID
9	Huckleberrys	7
10	Never Show Ups	22

> ❖ **说明**　如果你执行了前一章的查询 CH16_Add_Team，这个查询将删除 3 行，其中第 3 行表示的是球队 Aardvarks。

17.5　小结

本章首先简要讨论了用于从表中删除行的 DELETE 语句的语法，并列举了一个删除表中所有行的简单示例。接着简要地复习了事务，并介绍了数据库系统 Microsoft Access 如何使用事务来防范错误。

接下来讨论了如何使用 WHERE 子句来筛选要删除的行，介绍了如何使用 SELECT 语句来查看你打算删除的行，以及如何将这种 SELECT 语句转换为 DELETE 语句。

最后深入探索了如何使用子查询筛选要删除的行（根据其他表是否存在相关的行），并通过一些示例演示了如何编写 DELETE 查询。

下一节列出了多个你可独自解决的问题。

17.6　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 删除从未被订购过的商品。

  提示：需要先删除 Product_Vendors 表中的相关行，再删除 Products 表中的行。

  解决方案见 CH17_Delete_Products_Never_Ordered_1（删除了 4 行）和 CH17_Delete_Products_Never_Ordered_2（删除了 2 行）。请注意，如果你执行了前一章的查询 CH16_Add_Product，这里的第二个查询将删除 3 行。

  (2) 删除未售出任何商品的员工。

  解决方案见 CH17_Delete_Employees_No_Orders（删除了 1 行）。请注意，如果你执行了前一章的查询 CH16_Add_Employee，这里的查询将删除 2 行。

  (3) 删除不包含任何商品的类别。

  解决方案见 CH17_Delete_Categories_No_Products（删除了 1 行）。

- **Entertainment Agency 数据库**

  (1) 删除从未邀请演唱组合去演出的顾客。

  警告：要删除顾客，必须确保 Engagements 和 Musical_Preferences 表中都没有与之相关的行。

  解决方案见 CH17_Delete_Customers_Never_Booked1（删除了 5 行）和 CH17_Delete_Customers_Never_Booked2（删除了 2 行）。请注意，如果你执行了前一章的查询 CH16_Add_Customer，这里的第二个查询将删除 3 行。

  (2) 删除没有任何演唱组合演奏的音乐风格。

  警告：如果你只是查找没有出现在 Entertainer_Styles 表中的 StyleID 值，查询将失败，因为找出的有些风格还出现在了 Musical_Preferences 表中。

  解决方案见 CH17_Delete_Styles_No_Entertainer（删除了 5 行）。如果你执行了前一章的查询 CH16_Add_Style，这里的查询将删除 8 行。

  (3) 删除不属于任何演唱组合的成员。

  解决方案见 CH17_Delete_Members_Not_In_Group（没有删除任何行）。

- **School Scheduling 数据库**

  (1) 删除没有注册任何课程的学生。

  解决方案见 CH17_Delete_Students_No_Classes（删除了 1 行）。如果你执行了前一章的查询 CH16_Add_Student，将删除 2 行。

(2) 删除不包含任何课程的科目。

提示：需要分别从 Faculty_Subjects 和 Subjects 表中删除相关和满足条件的行。在第二个查询中，千万不要删除这样的科目，即其他包含课程的科目的先修科目。

解决方案见 CH17_Delete_Subjects_No_Classes_1（如果你执行了本章前面的查询 CH17_Delete_Classes_No_Studentss_2，将删除 64 行，否则将删除 8 行）和 CH17_Delete_Subjects_No_Classes_2（如果你执行了本章前面的查询 CH17_Delete_Classes_No_Students_2，将删除 33 行，否则将删除 4 行）。

- **Bowling League 数据库**

(1) 删除已复制到归档表中的联赛数据。

提示：需要从 Bowler_Scores、Match_Games、Tourney_Matches 和 Tournaments 表中删除行。除非你执行了第 16 章的 4 个归档查询，否则不会删除任何行。

解决方案见 CH17_Delete_Archived_2017_ Tournaments_1（删除了 1344 行）、CH17_Delete_Archived_2017_Tournaments_2（删除了 168 行）、CH17_Delete_Archived_2017_Tournaments_3（删除了 57 行）和 CH17_Delete_Archived_2017_Tournaments_4（删除了 14 行）。如果你的解决方案类似于 CH17_Delete_Archived_Tournaments_1_WRONG，那它就是错的。

(2) 删除没有举行的保龄球比赛场次。

警告：确保 Bowler_Scores 和 Match_Games 表中都没有匹配的行。

解决方案见 CH17_Delete_Matches_Not_Played（如果你执行了前一章的查询 CH16_Copy_2017_Tournaments，将删除 58 行，否则将删除 1 行）。

第六部分

解决棘手问题

第 18 章 否定型问题和多条件型问题

“每个复杂的问题都有简单、显而易见但错误的答案。”

——亨利·路易斯·门肯

本章涵盖如下主题：

- ❑ 简单地复习集合
- ❑ 解决否定型问题
- ❑ 根据多个肯定条件查找
- ❑ 语句举例
- ❑ 小结
- ❑ 练习

至此，你应该掌握了 SQL 数据库语言的基础知识（如果你是个好学生，完成了所有的示例语句和练习，更应如此）。现在该更上一层楼了。本章将带你处理更复杂的问题，它们有的是否定型的，有的是多条件型的；第 19 章将介绍如何在值表达式中使用 CASE 进行逻辑测试；第 20 章将引导你跳出固有的思维模式，使用不相关的（disconnected）表来解决问题；第 21 章将演示如何在对数据进行分组时计算小计；第 22 章将带你去领略相关数据的“窗口”。我们开始吧！

18.1 简单地复习集合

第 7 章使用了韦恩图来帮助你明白如下两点：解决多条件问题时需要获取两个集合的重叠部分；解决否定型问题时需要排除两个集合的某些部分。要求不满足一个条件的问题（如没有使用牛肉的菜品），解决起来很容易，但要求不满足多个条件的问题（没有使用牛肉、胡萝卜和洋葱的菜品），解决起来就有点难了。要求满足条件的问题（使用了奶酪的菜品）也是如此，需要满足两个条件时（使用了牛肉和洋葱的菜品），问题解决起来有点难；而需要满足 3 个甚至更多的条件时，问题解决起来将非常难。对于要求满足一个或多个条件，同时不满足多个条件的问题，解决起来更是难上加难。

18.1.1 满足多个条件的集合

先来看多条件情形，因为这更容易理解。图 18-1 显示了问题“使用了牛肉、洋葱和胡萝卜的菜品”的一种可能的解决方案。

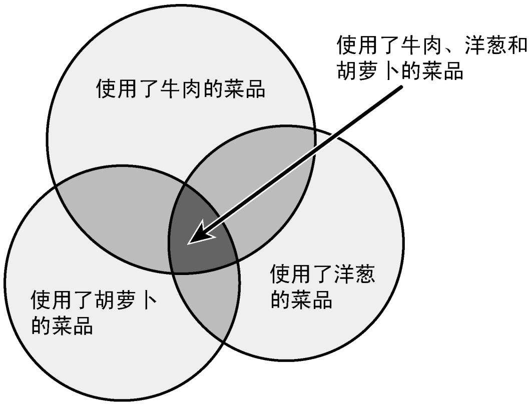


图 18-1　使用了牛肉、洋葱和胡萝卜的菜品

看起来非常简单，不是吗？但别忘了，菜品及其使用的食材本身都是包含多行的数据集（你能想出只使用一种食材的菜品吗？）。如果你试图使用类似于下面的查找条件来解决这个问题，得到的答案将是错误的。

```
WHERE Ingredient IN ('Beef', 'Onions', 'Carrots')
```

为什么呢？别忘了，数据库系统根据查找条件检查每一行。如果一种菜品使用了牛肉、洋葱或胡萝卜，它就满足上述查找条件。你要查找的是使用了全部三种而不是其中一种食材的菜品，为此需要使用更复杂的条件。有鉴于此，应像下面这样陈述这个问题：

“找出其食材列表中包含牛肉、包含洋葱且包含胡萝卜的菜品。”

第 8 章介绍了一种方法，可找出使用了两种食材的菜品，它在 FROM 子句中使用了两个不同的 SELECT 语句（CH08_Beef_And_Garlic_Recipes）；第 14 章介绍了另一种方法，它创造性地使用了 GROUP BY 和 HAVING（CH14_Beef_And_Garlic_Recipes）；本章将介绍解决这类问题的其他方法。

18.1.2　不满足多个条件的集合

要找出不满足多个条件的集合，需要分别找出满足各个条件的集合，再从总集合中剔除这些集合。如果要找出没有使用牛肉、洋葱和胡萝卜的菜品，可使用类似于图 18-2 所示的维恩图。

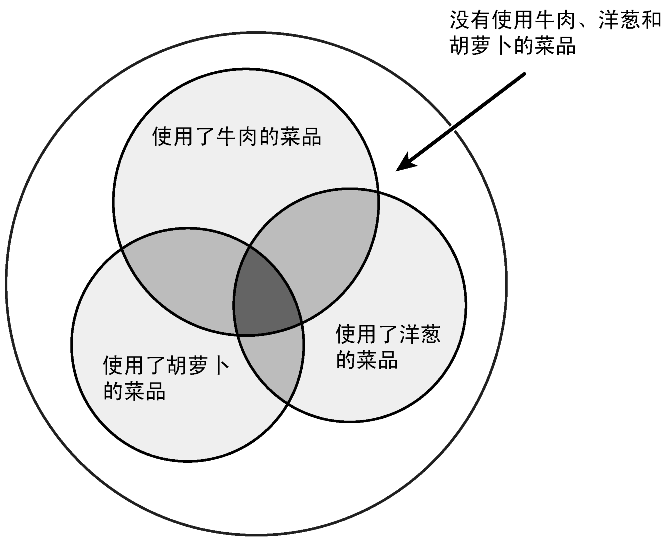


图 18-2　没有使用牛肉、洋葱和胡萝卜的菜品

这相当于找出所有使用了牛肉的菜品，将这些菜品从由所有菜品组成的集合中剔除，再找出所有使用了洋葱的菜品并将它们都剔除，然后找出所有使用了胡萝卜的菜品，并将它们都剔除。余下的菜品就是答案。同样，你可能想使用类似于下面的条件来解决这个问题：

```
WHERE Ingredient NOT IN ('Beef', 'Onions', 'Carrots')
```

这样做不可行，根据前面的讨论，你应该能够看出其中的原因。类似上面这样的查找条件将返回使用了除牛肉、洋葱和胡萝卜外其他任何食材的菜品，它找到并剔除只使用了这三种食材的菜品（这样的菜品真的很怪）。鉴于对于任何菜品，其使用的食材都构成一个集合，因此你需要像下面这样考虑这个问题：

"找出其食材列表中不包含牛肉、不包含洋葱且不包含胡萝卜的菜品。"

也可换种说法，这样陈述这个问题：

"找出其食材列表中包含牛肉、洋葱或胡萝卜的菜品，生成一个列表，再找出不在这个列表中的菜品。"

本书前面没有解决过这个问题，但本章一定会介绍一些解决方式。

18.1.3　满足一些条件同时不满足另一些条件的集合

出于完整性考虑，下面来简单地说说这样的情形：将满足一个或多个条件的包含在内，再将其中满足另外一个或多个条件的剔除。假设你要获取所有使用了牛肉的菜品——同时使用了洋葱或胡萝卜的菜品除外。图 18-3 说明了这个问题。

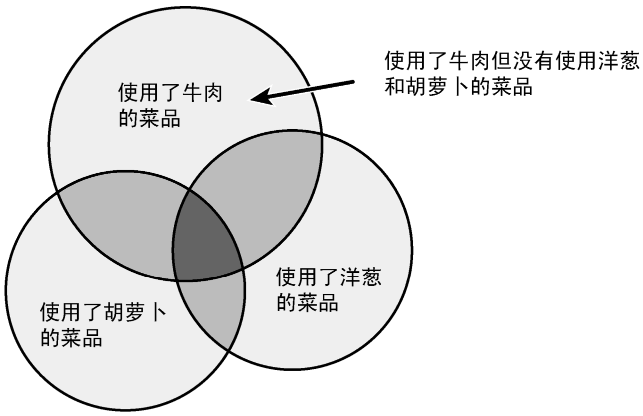


图 18-3　使用了牛肉但没有使用洋葱和胡萝卜的菜品

我们肯定，你能独自解决这个问题，但如果你试图像下面这样解决这个问题，就说明你没有真正搞明白：

```
WHERE Ingredient = 'Beef' AND Ingredient NOT IN ('Onions',
'Carrot')
```

这个查找条件无疑会找出所有使用了牛肉的菜品，但也会找出使用了除洋葱和胡萝卜外的其他食材的菜品（包括所有使用了牛肉的菜品）。再次声明，菜品使用的食材构成了一个集合，因此需要像下面这样解决这个问题：

"找出其食材列表中包含牛肉但不包含洋葱和胡萝卜的菜品。"

下面来介绍如何解决这些复杂的否定型多条件问题。

> ❖ **说明**　我将介绍多种解决否定型问题和多条件问题的方法，但不考虑性能。实际上，有些方法在一种数据库系统中的性能可能非常糟糕，但在另一种数据库系统中是最佳的解决方案。有关发现性能问题的高级方法，请参阅我与 Ben Clothier 和 Doug Steele 合著的 *Effective SQL: 61 Specific Ways to Write Better SQL*。话虽如此，我还是会指出在什么情况下，某种方法的性能可能比另一种方法高。

18.2　解决否定型问题

你可能还记得，你学习过如何解决简单的否定型问题。例如，第 9 章演示了如何找出未被任何菜品使用的食材（CH09_Ingredients_Not_Used）、如何找出没有订购头盔的顾客（CH09_Customers_No_Helmets）、如何找出没有签订任何演出合约的经纪人（CH09_Agents_No_Contracts），而第 11 章演示了如何找出从未退过课的学生（CH11_Students_Never_Withdrawn）以及如何找出未被订购过的商品（CH11_Products_Not_Ordered）。下面介绍如何使用 4 种不同的方法来解决否定型多条件问题：

- 外连接；
- NOT IN；
- NOT EXISTS；
- GROUP BY/HAVING。

> ❖ **说明**　贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”流程。考虑到你现在对这个流程已非常熟悉，为简化流程，我在下面的示例中都将转换和整理合二为一了。

18.2.1　使用外连接

第 9 章介绍过，可结合使用外连接和 IS NULL 测试来找出表中这样的行，即它在另一个表中没有匹配的行。一个这样的例子是找出未被任何菜品使用的食材。

“列出未在任何菜品中使用的食材。”

转换/整理	Select ingredient name from ~~the~~ ingredients ~~table~~ left outer join~~ed with the~~ recipe ingredients ~~table~~ on ingredients.ingredient ID ~~in the ingredients table matches~~ = recipe_ingredients.ingredient ID ~~in the recipe ingredients table~~ where recipe ID is null（基于食材 ID 相同将食材表左外连接到菜品食材表，从菜品 ID 为 Null 的行中选择食材名）

SQL

```sql
SELECT Ingredients.IngredientName
FROM Ingredients LEFT OUTER JOIN
     Recipe_Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID
WHERE Recipe_Ingredients.RecipeID IS NULL
```

CH09_Ingredients_Not_Used（20 行）

IngredientName
Halibut
Chicken, Fryer
Bacon
Iceberg Lettuce
Butterhead Lettuce
Scallop
Vinegar
Red Wine
<< 其他行 >>

请注意，这之所以可行，是因为如果第 2 个表中没有匹配的行，数据库引擎将把该表中的列值设置为 Null。可使用这种方法来将第 2 个表中满足特定条件的行排除在外，但必须结合使用第 9 章介绍的外连接和第 11 章介绍的子查询，因为必须先根据排除条件对第二个表进行筛选。

下面使用外连接和子查询来解决前面未使用牛肉、洋葱和胡萝卜的问题：

“找出未使用牛肉、洋葱和胡萝卜的菜品。”

转换/整理	Select recipe ID ~~and~~ recipe title from ~~the~~ recipes ~~table~~ left outer join~~ed with the~~ (selec~~tion of~~ recipe IDs from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on Recipe_Ingredients.recipe ID ~~in the recipe ingredients table matches~~ = Ingredients.recipe ID ~~in the ingredients table~~ where ingredient name ~~is~~ in ~~the values~~ (‘Beef’, ‘Onion’, ~~or~~ ‘Carrot’)) where ~~the~~ recipe ID ~~in the selection~~ is NULL ~~empty~~（基于菜品 ID 相同将菜品食材表内连接到食材表，从食材名为 Beef、Onion 或 Carrot 的行中选择菜品 ID，生成一个结果集；基于菜品 ID 相同将菜品表左外连接到上述结果集，在该结果集的菜品 ID 为 NULL 的行中，选择来自菜品表的菜品 ID 和菜品名）

SQL

```
SELECT Recipes.RecipeID, Recipes.RecipeTitle
FROM Recipes LEFT JOIN
   (SELECT Recipe_Ingredients.RecipeID
    FROM Recipe_Ingredients INNER JOIN Ingredients
      ON Recipe_Ingredients.IngredientID =
      Ingredients.IngredientID
    WHERE Ingredients.IngredientName
      IN ('Beef', 'Onion', 'Carrot'))
     AS RBeefCarrotOnion
  ON Recipes.RecipeID = RBeefCarrotOnion.RecipeID
WHERE RBeefCarrotOnion.RecipeID Is Null;
```

CH18_Recipes_NOT_Beef_Onion_Carrot_OUTERJOIN（8 行）

RecipeID	RecipeTitle
4	Garlic Green Beans
5	Fettuccini Alfredo
6	Pollo Picoso
7	Mike’s Summer Salad
8	Trifle
10	Yorkshire Pudding
12	Asparagus
15	Coupe Colonel

外连接右边的查询找出所有使用了牛肉、洋葱或胡萝卜的菜品的 ID，接下来，外连接和 IS NULL 测试将这些菜品 ID 都排除在外，只包含不与它们匹配的菜品。你可能想只在子查询中将菜品食材表内连接到食材表，而将检查牛肉、洋葱和胡萝卜的条件移到最后的 WHERE 子句中。然而，第 9 章说过，在左外连接的右边（或右外连接的左边）使用筛选器的作用是，让外连接的结果中包含列值为 Null 的行。如果你按刚才说的做，效果将与使用内连接一样，无法解决这个问题。

18.2.2　使用 NOT IN

第 11 章介绍了如何使用 NOT IN 来解决简单的否定型问题。例如，在示例数据库 Sales Orders 中，有一个找出从未订购过的商品的查询（CH11_Products_Not_Ordered）；在 Entertainment Agency 数据库中，有一个查询列出未签订任何演出合约的经纪人（CH11_Bad_Agents）。但你可能猜到了，查找不满足多个条件的行有点棘手。

下面使用 NOT IN 来解决 Recipes 数据库中的一个老问题。我们先采用笨办法，即使用三个不同的 NOT IN 子句。

“找出未使用牛肉、洋葱和胡萝卜的菜品。”

转换/整理	Select recipe ID ~~and~~ recipe title from ~~the~~ recipes ~~table~~ where ~~the~~ recipe ID is not in ~~the~~ (selec~~tion of~~ recipe IDs from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on Recipe_Ingredients.recipe ID ~~in the recipe ingredients table matches~~ = Ingredients.recipe ID ~~in the ingredients table~~ where ingredient name ~~is~~ = ‘Beef’) and ~~the~~ recipe ID ~~is~~ not in ~~the~~ (selec~~tion of~~ recipe IDs from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on Recipe_Ingredients.recipe ID ~~in the recipe ingredients table matches~~ = Ingredients.recipe ID ~~in the ingredients table~~ where ingredient name ~~is~~ = ‘Onion’) and ~~the~~ recipe ID is not in ~~the~~ (selec~~tion of~~ recipe IDs from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on Recipe_Ingredients.recipe ID ~~in the recipe ingredients table matches~~ = Ingredients.recipe ID ~~in the ingredients table~~ where ingredient name ~~is~~ = ‘Carrot’)

	（基于食材 ID 相同将菜品食材表内连接到食材表，从食材名为 Beef 的行中选择菜品 ID，生成第一个列表；基于食材 ID 相同将菜品食材表内连接到食材表，从食材名为 Onion 的行中选择菜品 ID，生成第二个列表；基于食材 ID 相同将菜品食材表内连接到食材表，从食材名为 Carrot 的行中选择菜品 ID，生成第三个列表；在菜品表中，从菜品 ID 不在第一个列表、第二个列表和第三个列表中的行中选择菜品 ID 和菜品名）

SQL

```
SELECT Recipes.RecipeID, Recipes.RecipeTitle
FROM Recipes
WHERE Recipes.RecipeID NOT IN
   (SELECT Recipe_Ingredients.RecipeID
    FROM Recipe_Ingredients
      INNER JOIN Ingredients
      ON Recipe_Ingredients.IngredientID =
      Ingredients.IngredientID
    WHERE Ingredients.IngredientName = 'Beef')
  AND Recipes.RecipeID NOT IN
   (SELECT Recipe_Ingredients.RecipeID
    FROM Recipe_Ingredients
      INNER JOIN Ingredients
      ON Recipe_Ingredients.IngredientID =
      Ingredients.IngredientID
    WHERE Ingredients.IngredientName = 'Onion')
  AND Recipes.RecipeID NOT IN
   (SELECT Recipe_Ingredients.RecipeID
    FROM Recipe_Ingredients
      INNER JOIN Ingredients
      ON Recipe_Ingredients.IngredientID =
      Ingredients.IngredientID
    WHERE Ingredients.IngredientName = 'Carrot');
```

这个查询执行了 3 次消除操作，依次消除所有使用了牛肉的菜品、所有使用了洋葱的菜品和所有使用了胡萝卜的菜品（我保存了这个查询，并将其命名为 CH18_Recipes_NOT_Beef_Onion_Carrot_NOTIN_1）。但只要想一想就知道，如果能够在一个子查询中搜集所有使用了牛肉、洋葱或胡萝卜的菜品，就只需执行一次消除，如下所示。

“找出未使用牛肉、洋葱和胡萝卜的菜品。”

转换/整理	Select recipe ID ~~and~~ recipe title from ~~the~~ recipes ~~table~~ where ~~the~~ recipe ID is not in ~~the~~ (selec~~tion of~~ recipe IDs from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on Recipe_Ingredients.recipe ID ~~in the recipe ingredients table matches~~ = Ingredients.recipe ID ~~in the ingredients table~~ where ingredient name ~~is~~ in ~~the values~~ (‘Beef’, ‘Onion’, ~~or~~ ‘Carrots’))（基于菜品 ID 相同将菜品食材表内连接到食材表，从食材名为 Beef、Onion 或 Carrots 的行中选择菜品 ID，生成一个列表；在菜品表中，从菜品 ID 不在上述列表中的行中选择菜品 ID 和菜品名）

SQL

```
SELECT Recipes.RecipeID, Recipes.RecipeTitle
FROM Recipes
WHERE Recipes.RecipeID NOT IN
  (SELECT Recipe_Ingredients.RecipeID
   FROM Recipe_Ingredients
   INNER JOIN Ingredients
     ON Recipe_Ingredients.IngredientID =
      Ingredients.IngredientID
   WHERE Ingredients.IngredientName
     IN ('Beef', 'Onion', 'Carrot'));
```

这个查询简单得多，实际上，这也是解决否定型多条件问题的最简单方式。这个查询的效率也非常高，因为数据库系统只需运行子查询一次，再根据其结果将匹配的行排除在外。我将这个查询保存到了示例数据库 Recipes 中，并将其命名为 CH18_Recipes_NOT_Beef_Onion_Carrot_NOTIN_2。

18.2.3 使用 NOT EXISTS

第 11 章还介绍了如何使用 EXISTS 和子查询根据单个条件查找相关的数据。可以想见，可通过扩展这种方法来处理多个条件。处理否定条件也很简单，只需使用 NOT EXISTS 即可。下面使用 NOT EXISTS 来解决“未使用牛肉、洋葱和胡萝卜”的问题。

“找出未使用牛肉、洋葱和胡萝卜的菜品。”

转换/整理　Select recipe ID ~~and~~ recipe title from ~~the~~ recipes ~~table~~ where ~~does~~ not exist ~~the~~ (selec~~tion of~~ recipe ID~~s~~ from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on Recipe_Ingredients.recipe ID ~~in the recipe ingredients table matches~~ = Ingredients.recipe ID ~~in the ingredients table~~ where ingredient name ~~is~~ in ~~the values~~ (‘Beef’, ‘Onion’, ~~or~~ ‘Carrot’) and ~~the~~ Recipe_Ingredients.recipe ID ~~from the recipe ingredients table matches the~~ = Recipes.recipe ID ~~from the recipes table~~)（对于菜品表中的每一行，做如下处理：基于菜品 ID 相同将菜品食材表内连接到食材表，从食材名为 Beef、Onion 或 Carrot 且菜品 ID 与该行的菜品 ID 相同的行中选择菜品 ID，如果生成的列表为空，就从该行选择菜品 ID 和菜品名）

SQL

```
SELECT Recipes.RecipeID, Recipes.RecipeTitle
FROM Recipes
WHERE NOT EXISTS
   (SELECT Recipe_Ingredients.RecipeID
    FROM Recipe_Ingredients
      INNER JOIN Ingredients
        ON Recipe_Ingredients.IngredientID =
           Ingredients.IngredientID
    WHERE Ingredients.IngredientName
        IN ('Beef', 'Onion', 'Carrot')
      AND Recipe_Ingredients.RecipeID =
          Recipes.RecipeID);
```

这个查询的工作原理与使用 NOT IN 的查询类似。首先获取使用了牛肉、洋葱或胡萝卜的菜品，再通过匹配菜品 ID 和使用 NOT EXISTS 将它们排除在外。这种方法的一个缺点是，在子查询中必须引用主查询中的一个列，这意味着数据库系统必须为 Recipes 表中的每行执行子查询一次，即每个不同的 RecipeID 值一次（在一些进阶书中，这种子查询被称为关联子查询，因为它实际上是与外部查询中的各行相关联的）。我将这个查询保存到了示例数据库 Recipes 中，并将其命名为 CH18_Recipes_NOT_Beef_Onion_Carrot_NOTEXISTS。

18.2.4　使用 GROUP BY/HAVING

第 14 章介绍了如何判断是否至少有 *n* 行满足一个或多个条件，例如，在 Entertainment Agency 数据库中，有一个查询列出演奏爵士乐且成员至少有 3 人的演唱组合（CH14_Jazz_Entertainers_More_Than_3）。

可否通过检查计数是否为零来找出不符合条件的数据集？当然可以！下面使用 GROUP BY 和 HAVING COUNT = 0 来解决“未使用牛肉、洋葱和胡萝卜”的问题。

“找出未使用牛肉、洋葱和胡萝卜的菜品。”

转换/整理　Select recipe ID ~~and~~ recipe title from ~~the~~ recipes ~~table~~ left join~~ed with the~~ (selec~~tion of~~ recipe ID from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients ~~table~~ on Ingredients.ingredient ID ~~in the ingredients table equals matches~~ = Recipe_Ingredients.ingredient ID ~~in the recipe ingredients table~~ where ingredient name ~~is~~ in ~~the values~~ (‘beef’, ‘onion’, ~~or~~ ‘carrot’)) AS RIBOC on Recipes.recipe ID ~~in the recipes table equals matches~~ = RIBOC. recipe ID ~~in the selection~~ where RIBOC.recipe ID ~~in the selection~~ is NULL ~~empty, then~~ group~~ed~~ by recipe ID ~~and~~ recipe title ~~and~~ having ~~the~~ count ~~of~~ RIBOC.recipeID ~~in the selection equals zero~~ = 0（基于菜品 ID 相同将菜品食材表内连接到食材表，从食材名为 beef、onion 或 carrot 的行中选择菜品 ID，将生成的结果集命名为 RIBOC；基于菜品 ID 相同将菜品表左外连接到 RIBOC，按菜品 ID 和菜品名分组；对于各个分组，如果其中不为 Null 的 RIBOC.recipeID 值为零个，就返回来自菜品表的菜品 ID 和菜品名称）

SQL

```
SELECT Recipes.RecipeID, Recipes.RecipeTitle
FROM Recipes LEFT JOIN
   (SELECT Recipe_Ingredients.RecipeID
    FROM Recipe_Ingredients
      INNER JOIN Ingredients
        ON Ingredients.IngredientID =
           Recipe_Ingredients.IngredientID
    WHERE Ingredients.IngredientName
      IN ('Beef', 'Onion', 'Carrot')) AS RIBOC
  ON Recipes.RecipeID = RIBOC.RecipeID
WHERE RIBOC.RecipeID IS NULL
GROUP BY Recipes.RecipeID, Recipes.RecipeTitle
HAVING COUNT(RIBOC.RecipeID) = 0;
```

如果你注意到这很像使用左外连接的解决方案，那么你完全正确。事实上，只需外连接到一个表时，使用左外连接的解决方案更好，因为它没有将行分组的开销。但进行排除时，如果需要连接多个表并使用其他条件，那么使用

GROUP BY 和 COUNT 是个不错的选择。第 13 章介绍过，使用基于列的 COUNT（或其他聚合函数）时，如果该列的值为 Null，聚合函数将忽略相应的行。这就是左外连接的子查询结果中没有匹配的行时，COUNT(RIBOC.RecipeID)管用的原因：对于给定的菜品，如果使用了牛肉、洋葱或胡萝卜的菜品集中没有与之匹配的行，COUNT(RIBOC.RecipeID)的结果将为零。这个查询保存到了示例数据库 Recipes 中，名为 CH18_Recipes_NOT_Beef_Onion_Carrot_GROUPBY。

我们来看一个使用 GROUP BY 和 HAVING 更合适的例子：

“找出使用了黄油但未使用牛肉、洋葱和胡萝卜的菜品。”

转换/整理

Select recipe ID ~~and~~ recipe title from ~~the~~ recipes ~~table~~ inner join~~ed with the~~ recipe ingredients ~~table on~~ Recipes.recipe ID ~~in the recipes table equals~~ = Recipe_ Ingredients.recipe ID ~~in the recipe ingredients table, then~~ inner join~~ed with the~~ ingredients ~~table~~ on Ingredients.ingredient ID ~~in the ingredients table equals~~ = Recipe_Ingredients.ingredient ID ~~in the recipe ingredients table, then~~ left join~~ed with the~~ (select~~ion of~~ recipe ID from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ ingredients table on Ingredients.ingredient ID ~~in the ingredients table equals~~ = Recipe_Ingredients.ingredient ID ~~in the recipe ingredients table~~ where ingredient name ~~is~~ in ~~the values~~ (‘beef’, ‘onion’, ~~or~~ ‘carrot’)) AS RIBOC on Recipes. recipe ID ~~in the recipes table equals~~ RIBOC.recipe ID ~~in the selection~~ where ingredient name ~~in the ingredients table equals~~ = ‘Butter’ and RIBOC.recipe ID ~~in the selection~~ is NULL ~~empty, then~~ group~~ed~~ by recipe ID ~~and~~ recipe title ~~and~~ having ~~the~~ count ~~of~~ RIBOC.recipe ID ~~in the selection equals zero~~ = 0（基于菜品 ID 相同将菜品食材表内连接到食材表，从食材名为 beef、onion 或 carrot 的行中选择菜品 ID，将生成的结果集命名为 RIBOC；依次基于菜品 ID 相同将菜品表内连接到菜品食材表、基于食材 ID 相同内连接到食材表，再基于菜品 ID 相同左外连接到结果集 RIBOC；选择食材名为 Butter 且 RIBOC.recipe ID 为 Null 的行，再按来自菜品表的菜品 ID 和菜品名分组；对于每个分组，如果它包含的 RIBOC.recipe ID 不为 Null 的行数为零，就选择其来自菜品表的菜品 ID 和菜品名）

SQL

```
SELECT Recipes.RecipeID, Recipes.RecipeTitle
FROM ((Recipes INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID
      = Recipe_Ingredients.RecipeID)
INNER JOIN Ingredients
  ON Ingredients.IngredientID =
     Recipe_Ingredients.IngredientID)
LEFT JOIN
   (SELECT Recipe_Ingredients.RecipeID
    FROM Recipe_Ingredients
      INNER JOIN Ingredients
        ON Ingredients.IngredientID =
          Recipe_Ingredients.IngredientID
    WHERE Ingredients.IngredientName IN
      ('Beef', 'Onion', 'Carrot')) AS RIBOC
  ON Recipes.RecipeID=RIBOC.RecipeID
WHERE Ingredients.IngredientName = 'Butter'
  AND RIBOC.RecipeID IS NULL
GROUP BY Recipes.RecipeID, Recipes.RecipeTitle
HAVING COUNT(RIBOC.RecipeID) = 0;
```

CH18_Recipes_Butter_NOT_Beef_Onion_Carrot_GROUPBY（2 行）

RecipeID	RecipeTitle
5	Fettuccini Alfredo
12	Asparagus

在这个示例中，使用 GROUP BY 和 HAVING 更合理，因为连接 Recipes、Recipe_Ingredients 和 Ingredients 表时，对于每个菜品，将返回多行，但我希望在最终的结果中，每个菜品只有一行。使用 GROUP BY 实现了为每个菜品返回一行的目标，而使用 HAVING 消除了所有使用了食材牛肉、洋葱或胡萝卜的菜品。

至此几乎介绍了解决否定型多条件问题的所有方法，本章后面将列举其他一些示例，还将提供一些需要你自己动手去解决的问题。

18.3 根据多个肯定条件查找

下面来看看硬币的另一面：根据多个条件进行查找的查询。你在第 8 章编写过这样的查询：如何在数据库 Sales Orders 中找出订购了自行车和头盔的顾客（CH08_Customers_Both_Bikes_And_Helmets）；如何在数据库 Entertainment

Agency 中找出给顾客 Berg 和 Hallmark 都演出过的演唱组合（CH08_Entertainers_Berg_AND_Hallmark）。下面更深入地探索众多解决这类问题的方法：

- ❑ 内连接；
- ❑ IN；
- ❑ EXISTS；
- ❑ GROUP BY/HAVING。

18.3.1　使用内连接

第 7 章介绍过，要找出两个集合中匹配的元素，可计算它们的交集。我还指出过，要在 SQL 中执行基于键值的交集运算，最常见的做法是使用内连接。主键唯一地标识了表中的每行，因此基于主键值的交集运算将返回两个集合中都有的行。

因此，要找出满足多个条件的行，一种方法是对于每个条件，都使用一个子查询来创建一个数据集，再基于主键值将这些数据集连接起来。来看一个来自数据库 Entertainment Agency 的示例：

“列出邀请演唱组合 Carol Peacock Trio、Caroline Coie Cuartet 和 Jazz Persuasion 出演过的顾客。”

转换/整理	Select ~~the unique~~ DISTINCT CPT.customer ID, CPT.cust~~omer~~ first name, ~~and~~ CPT.cust~~omer~~ last name from ~~the~~ (select~~ion of~~ customer id, cust~~omer~~ first name, cust~~omer~~ last name from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on Customers.customer ID ~~in the customers table equals~~ = Engagements.customer ID ~~in the engagements table then~~ inner join~~ed with the~~ entertainers ~~table~~ on Engagements.entertainer ID ~~in the engagements table equals~~ = Entertainers.entertainer ID ~~in the entertainers table~~ where Entertainers.ent~~ertainer~~ stage name ~~in the entertainers table equals~~ = ‘Carol Peacock Trio’) AS CPT inner join~~ed with the~~ (select~~ion of~~ customer id from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on Customers.customer ID ~~in the customers table equals~~ = Engagements.customer ID ~~in the engagements table then~~ inner join~~ed with the~~ entertainers ~~table~~ on Engagements.entertainer ID ~~in the engagements table equals~~ = Entertainers.entertainer ID ~~in the entertainers table~~ where Entertainers.ent~~ertainer~~ stage name ~~in the entertainers table equals~~ = ‘Caroline Coie Cuartet’) AS CCC on CPT.customer ID ~~in the first selection equals~~ = CCC.customer ID ~~in the second selection~~ inner join~~ed with the~~ (select~~ion of~~ customer id from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on Customers.customer ID ~~in the customers table equals~~ = Engagements.customer ID ~~in the engagements table then~~ inner join~~ed with the~~ entertainers ~~table~~ on Engagements.entertainer ID ~~in the engagements table equals~~ = Entertainers.entertainer ID ~~in the entertainers table~~ where Entertainers.ent~~ertainer~~ stage name ~~in the entertainers table equals~~ = ‘Jazz Persuasion’) AS JP on CCC.customer ID ~~in the second selection equals~~ = JP.customer ID ~~in the third selection~~ （依次基于顾客 ID 相同将顾客表内连接到演出合约表、基于演唱组合 ID 相同内连接到演唱组合表，从演唱组合艺名为 Carol Peacock Trio 的行中选择顾客的 ID、名和姓，将生成的结果集命名为 CPT；依次基于顾客 ID 相同将顾客表内连接到演出合约表、基于演唱组合 ID 相同内连接到演唱组合表，从演唱组合艺名为 Caroline Coie Cuartet 的行中选择顾客的 ID、名和姓，将生成的结果集命名为 CCC；依次基于顾客 ID 相同将顾客表内连接到演出合约表、基于演唱组合 ID 相同内连接到演唱组合表，从演唱组合艺名为 Jazz Persuasion 的行中选择顾客的 ID、名和姓，将生成的结果集命名为 JP；依次基于顾客 ID 相同将结果集内连接到结果集 CCC、基于顾客 ID 相同内连接到结果集 JP，从生成的结果集中选择来自结果集 CPT 的不同顾客 ID、顾客名和顾客姓）

SQL

```
SELECT DISTINCT CPT.CustomerID, CPT.CustFirstName,
  CPT.CustLastName
FROM ((SELECT Customers.CustomerID,
  Customers.CustFirstName, Customers.CustLastName
       FROM (Customers INNER JOIN Engagements
         ON Customers.CustomerID =
            Engagements.CustomerID)
       INNER JOIN Entertainers
         ON Engagements.EntertainerID =
            Entertainers.EntertainerID
       WHERE Entertainers.EntStageName =
        'Carol Peacock Trio') As CPT
  INNER JOIN
      (SELECT Customers.CustomerID
       FROM (Customers INNER JOIN Engagements
         ON Customers.CustomerID =
            Engagements.CustomerID)
       INNER JOIN Entertainers
         ON Engagements.EntertainerID =
            Entertainers.EntertainerID
```

```
                    WHERE Entertainers.EntStageName =
                     'Caroline Coie Cuartet') As CCC
                ON CPT.CustomerID = CCC.CustomerID)
              INNER JOIN
                   (SELECT Customers.CustomerID
                    FROM (Customers INNER JOIN Engagements
                      ON Customers.CustomerID =
                         Engagements.CustomerID)
                    INNER JOIN Entertainers
                      ON Engagements.EntertainerID =
                         Entertainers.EntertainerID
                    WHERE Entertainers.EntStageName =
                     'Jazz Persuasion') As JP
                ON CCC.CustomerID = JP.CustomerID;
```

CH18_Customers_Peacock_Coie_Jazz_INNERJOIN（2 行）

CustomerID	CustFirstName	CustLastName
10004	Dean	McCrae
10010	Zachary	Ehrlich

FROM 子句中的 3 个 SELECT 表达式获取所需的 3 个集合：邀请 Carol Peacock Trio 出演过的顾客；邀请 Caroline Coie Cuartet 出演过的顾客；邀请 Jazz Persuasion 出演过的顾客。在第一个子查询中，我选择了包含顾客姓名的列，以便在最终结果中显示它们，但在第二个和第三个子查询中，只需选择 CustomerID（Customers 表的主键），以便通过连接找出邀请这三个演唱组合出演过的所有顾客。最后，我使用关键字 DISTINCT 消除重复的行（如果顾客多次邀请某个演唱组合出演过，就会有重复的行）。

如果你回过头去看第 8 章，将发现这里使用的方法与查询 CH08_Entertainers_Berg_AND_Hallmark 相同。唯一的差别是，这里在外部 SELECT 语句（而不是每个子查询）中使用了 DISTINCT。

18.3.2 使用 IN

下面使用 IN 来解决邀请 3 个演唱组合出演的顾客的问题。使用 IN 根据多个条件查找时，你可能想采取下面这种简单方式：

```
SELECT Customers.CustomerID, Customers.CustFirstName,
  Customers.CustLastName
FROM Customers
WHERE Customers.CustomerID IN
   (SELECT Customers.CustomerID
    FROM (Customers INNER JOIN Engagements
      ON Customers.CustomerID = Engagements.CustomerID)
    INNER JOIN Entertainers
      ON Engagements.EntertainerID = Entertainers.
         EntertainerID
WHERE Entertainers.EntStageName IN
  ('Carol Peacock Trio', 'Caroline Coie Cuartet',
       'Jazz Persuasion'))
```

这行不通，为什么呢？答案是将得到邀请了这 3 个演唱组合中任何一个的所有顾客，而不是邀请了所有 3 个演唱组合的顾客！这个查询保存在示例数据库 Entertainment Agency 中，名为 CH18_Customers_ Peacock_Coie_Jazz_IN_WRONG。

别忘了，要查找邀请了所有 3 个演唱组合的顾客，需要计算 3 个集合的交集：邀请了 Carol Peacock Trio 的顾客集；邀请了 Caroline Coie Cuartet 的顾客集；邀请了 Jazz Persuasion 的顾客集。要使用 IN 解决这个问题，需要 3 个 IN 子句，且必须找出同时位于这 3 个集合中的顾客。我们来尝试这样做：

“列出邀请演唱组合 Carol Peacock Trio、Caroline Coie Cuartet 和 Jazz Persuasion 出演过的顾客。”

转换/整理

Select customer ID, cust~~omer~~ first name, ~~and~~ cust~~omer~~ last name from ~~the~~ customers ~~table~~ where customerID ~~is~~ in ~~the~~ (selecti~~on of~~ customer id from ~~the~~ engagements ~~table~~ inner join~~ed with the~~ entertainers ~~table~~ on Engagements.entertainer ID ~~in the engagements table equals~~ = Entertainers.entertainer ID ~~in the entertainers table~~ where Entertainers.ent~~ertainer~~ stage name ~~in the entertainers table equals~~ = 'Carol Peacock Trio') and customer id ~~is~~ in ~~the~~ (selecti~~on of~~ customer id from ~~the~~ engagements ~~table~~ inner join~~ed with the~~ entertainers ~~table~~ on Engagements.entertainer ID ~~in the engagements table equals~~ = Entertainers.entertainer ID ~~in the entertainers table~~ where Entertainers.ent~~ertainer~~ stage name ~~in the entertainers table equals~~ = 'Caroline Coie Cuartet') and customer id ~~is~~ in ~~the~~ (selecti~~on of~~ customer id from ~~the~~ engagements ~~table~~ inner join~~ed with the~~ entertainers ~~table~~ on Engagements.entertainer ID ~~in the engagements table equals~~ = Entertainers.entertainer ID ~~in the entertainers table~~ where Entertainers.ent~~ertainer~~ stage name ~~in the entertainers table equals~~ = 'Jazz Persuasion')（基于演唱组合 ID 相同将演出合约表内连接到演唱组合表，从演唱组合艺名为 Carol Peacock Trio 的行中选择顾客 ID，生成第一个列表；基于演唱组合 ID 相同将演出合约表内连接到演唱组合表，从演唱组合艺名为 Caroline Coie Cuartet 的行中选择顾客 ID，生成第二个列表；基于演唱组合 ID 相同将演出合约表内连接到演唱组合表，从演唱组合艺名为 Jazz Persuasion 的行中选择顾客 ID，生成第三个列表；在顾客表中，从顾客 ID 位于第一个列表、第二个列表和第三个列表中的行中选择顾客的 ID、名和姓）

SQL

```
SELECT Customers.CustomerID,
   Customers.CustFirstName,
   Customers.CustLastName
FROM Customers
WHERE Customers.CustomerID IN
    (SELECT Engagements.CustomerID
     FROM Engagements INNER JOIN Entertainers
       ON Engagements.EntertainerID =
          Entertainers.EntertainerID
     WHERE Entertainers.EntStageName =
       'Carol Peacock Trio')
AND Customers.CustomerID IN
    (SELECT Engagements.CustomerID
     FROM Engagements INNER JOIN Entertainers
       ON Engagements.EntertainerID =
          Entertainers.EntertainerID
     WHERE Entertainers.EntStageName =
       'Caroline Coie Cuartet')
AND Customers.CustomerID IN
    (SELECT Engagements.CustomerID
     FROM Engagements INNER JOIN Entertainers
       ON Engagements.EntertainerID =
          Entertainers.EntertainerID
     WHERE Entertainers.EntStageName =
       'Jazz Persuasion');
```

与使用内连接的查询一样，这个查询也返回两行。在前面的 SQL 中，我故意分开了其中的 3 个子查询，让你能够清楚如何获取 3 个集合。这个查询保存到了示例数据库 Entertainment Agency 中，名为 CH18_Customers_Peacock_Coie_Jazz_IN_RIGHT。

18.3.3 使用 EXISTS

为了使用 EXISTS 解决邀请 3 个演唱组合的顾客的问题，将使用的方法与使用 IN 解决这个问题时采用的方法类似。最重要的不同在于，每个子查询也必须匹配顾客 ID。由于你要检查每个集合是否为空，因此每个集合都必须与当前行的顾客 ID 匹配。下面演示了如何使用 EXISTS 解决这个问题：

"列出邀请演唱组合 Carol Peacock Trio、Caroline Coie Cuartet 和 Jazz Persuasion 出演过的顾客。"

转换/整理

Select customer ID, cust~~omer~~ first name, ~~and~~ customer last name from ~~the~~ customers ~~table~~ where ~~there~~ exists ~~the~~ (selecti~~on of~~ customer id from ~~the~~ engagements ~~table~~ inner join~~ed with the~~ entertainers ~~table~~ on Engagements.entertainer ID ~~in the engagements table equals~~ = Entertainers.entertainer ID ~~in the entertainers table~~ where Entertainers.ent~~ertainer~~ stage name ~~in the entertainers table equals~~ = 'Carol Peacock Trio' and ~~the~~ Engagements.customer ID ~~in the engagements table equals the~~ = Customers.customer ID ~~in the customers table~~) and ~~there~~ exists ~~the~~ (selection of customer id from ~~the~~ engagements ~~table~~ inner join~~ed with the~~ entertainers ~~table~~ on Engagements.entertainer ID ~~in the engagements table equals~~ = Entertainers.entertainer ID ~~in the entertainers table~~ where Entertainers.ent~~ertainer~~ stage name ~~in the entertainers table equals~~ = 'Caroline Coie Cuartet' and ~~the~~ Engagements.customer ID ~~in the engagements table equals the~~ = Customers.customer ID ~~in the customers table~~) and ~~there~~ exists ~~the~~ (selecti~~on of~~ customer id from ~~the~~ engagements ~~table~~ inner join~~ed with the~~ entertainers ~~table~~ on Engagements.entertainer ID ~~in the engagements table~~ equals = Entertainers.ent~~ertainer~~ ID ~~in the entertainers table~~ where Entertainers.ent~~ertainer~~ stage name ~~in the entertainers table equals~~ = 'Jazz Persuasion' and ~~the~~ Engagements.customer ID ~~in the engagements table equals the~~ = Customers.customer ID ~~in the~~

	~~customers table~~)（在顾客表中，对于其中的每行都做如下处理：基于演唱组合 ID 相同将演出合约表内连接到演唱组合表，从演唱组合艺名为 Caroline Coie Cuartet 且顾客 ID 与该行的顾客 ID 相同的行中选择顾客 ID，生成第一个结果集；基于演唱组合 ID 相同将演出合约表内连接到演唱组合表，从演唱组合艺名为 Caroline Coie Cuartet 且顾客 ID 与该行的顾客 ID 相同的行中选择顾客 ID，生成第二个结果集；基于演唱组合 ID 相同将演出合约表内连接到演唱组合表，从演唱组合艺名为 Jazz Persuasion 且顾客 ID 与该行的顾客 ID 相同的行中选择顾客 ID，生成第三个结果集；如果这三个结果集都不为空，就从顾客表的当前行中选择顾客的 ID、名和姓）

SQL

```
SELECT Customers.CustomerID,
  Customers.CustFirstName, Customers.CustLastName
FROM Customers
WHERE EXISTS
  (SELECT Engagements.CustomerID
   FROM Engagements INNER JOIN Entertainers
     ON Engagements.EntertainerID =
        Entertainers.EntertainerID
   WHERE Entertainers.EntStageName =
          'Carol Peacock Trio'
     AND Engagements.CustomerID =
         Customers.CustomerID)
AND EXISTS
  (SELECT Engagements.CustomerID
   FROM Engagements INNER JOIN Entertainers
     ON Engagements.EntertainerID =
        Entertainers.EntertainerID
   WHERE Entertainers.EntStageName =
         'Caroline Coie Cuartet'
     AND Engagements.CustomerID =
         Customers.CustomerID)
AND EXISTS
  (SELECT Engagements.CustomerID
   FROM Engagements INNER JOIN Entertainers
     ON Engagements.EntertainerID =
        Entertainers.EntertainerID
   WHERE Entertainers.EntStageName =
        'Jazz Persuasion'
   AND Engagements.CustomerID =
       Customers.CustomerID);
```

这个查询的工作原理与使用 IN 的查询类似。你找出 3 个顾客集（每个集合中的顾客都邀请了特定的演唱组合），并使用 EXISTS 进行测试。这种方法的一个缺点是，在子查询中必须引用主查询中的一个列，这意味着数据库系统必须为 Customers 表的每行执行每个子查询一次，即每个不同的 CustomerID 值一次（在有些高级图书中，这种子查询被称为关联子查询，因为它实际上是与外部查询中的每行相关联的）。这个子查询保存到了示例数据库 Entertainment Agency 中，名为 CH18_Customers_Peacock_Coie_Jazz_EXISTS。

18.3.4　使用 GROUP BY/HAVING

你可以尝试使用 GROUP BY 和 HAVING 来解决这个邀请了 3 个演唱组合的顾客的问题，但很难。本章前面使用 GROUP BY 和 HAVING 来解决与菜品和食材相关的问题时，我知道任何一种食材都只会在 Recipe_Ingredients 表中出现一次，但对顾客和演唱组合来说，情况并非如此，因为一位顾客可能多次邀请同一个演唱组合。诚然，我可使用 SELECT DISTINCT 并对其返回的结果进行分组，但既然有多种解决这个问题的方式，为何要自找麻烦呢?

考虑到这一点，下面来看示例数据库 Entertainment Agency 中一个有趣的问题。对这个问题来说，使用 GROUP BY 和 HAVING 来解决确实是最佳的选择。这个问题如下：

> “显示这样的顾客和演唱组合，即顾客喜欢的所有音乐风格都是演唱组合演奏的音乐风格。”

这是一个“多匹配”问题，因为每位顾客都可能告诉经纪人，有多种音乐风格是他喜欢的。这里的难点在于，“多”并非一个固定的数字，即需要匹配的音乐风格数可能因顾客而异。

来看看为了解决这个问题需要用到的表。图 18-4 列出了为找出所有的演唱组合及其演奏的音乐风格需要用到的表，而图 18-5 列出了为找出所有的顾客及其喜欢的音乐风格需要用到的表。

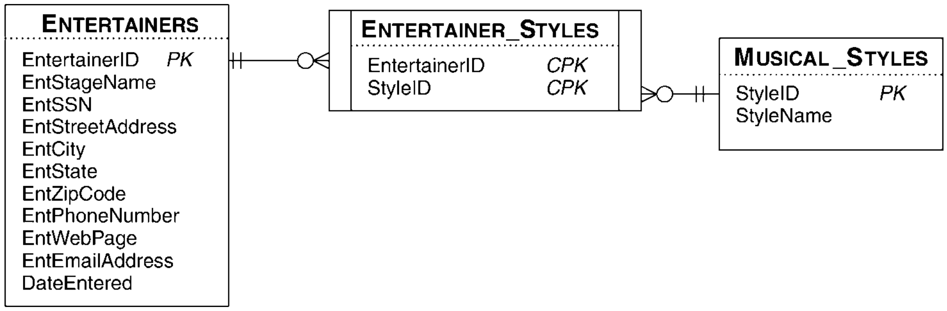


图 18-4　为列出所有的演唱组合及其演奏的音乐风格需要用到的表

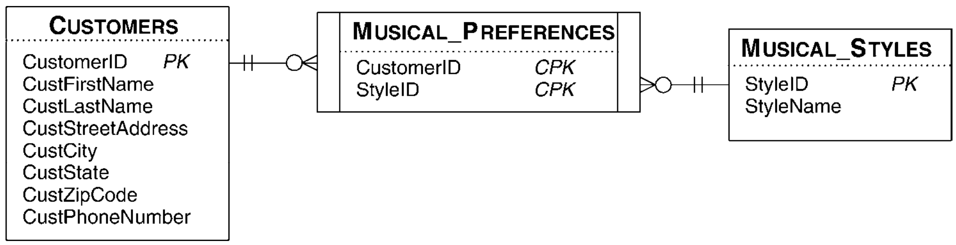


图 18-5　为列出所有的顾客及其喜欢的音乐风格需要用到的表

你发现在这两组表中都有的列了吗？StyleID 是不是呢？实际上，可能根本不需要 Musical_Styles 表——除非也要列出匹配的音乐风格。如果你查看示例数据库 Entertainment Agency 的完整结构图，将发现 Musical_Preferences 表的 StyleID 列和 Entertainer_Styles 表的 StyleID 列没有直接关系，但基于 StyleID 将这两个表连接起来完全合法，因为这两个表的 StyleID 列的数据类型相同。这样连接也合乎逻辑，因为连接用到的两个列的含义相同。你要找出顾客和演唱组合之间匹配的所有音乐风格，具体地说，你要找出这样的顾客和演唱组合：它们之间匹配的音乐风格数与顾客喜欢的音乐风格数相等。下面来解决这个问题：

“显示这样的顾客和演唱组合，即顾客喜欢的所有音乐风格都是演唱组合演奏的音乐风格。”

转换/整理

Select customer ID, cust~~omer~~ first name, cust~~omer~~ last name, entertainer ID, ent~~ertainer~~ stage name, ~~and the~~ count ~~of~~ (style ID) from ~~the~~ customers ~~table~~ inner join~~ed with the~~ musical preferences ~~table~~ on Customers.customer ID ~~in the customers table matches~~ = Musical_Styles.customer ID ~~in the musical styles table then~~ inner join~~ed with the~~ entertainer styles ~~table~~ on Musical_Styles.style ID ~~in the musical styles table matches~~ = Entertainer_Styles.style ID ~~in the entertainer styles table and finally~~ inner join~~ed with the~~ entertainers ~~table~~ on Entertainers.entertainer ID ~~in the entertainers table matches~~ = Entertainer_Styles.entertainer ID ~~in the entertainer styles table~~ grouped by customer ID, cust~~omer~~ first name, cust~~omer~~ last name, entertainer ID, ~~and~~ ent~~ertainer~~ stage name ~~and~~ having ~~the~~ count ~~of~~ (style ID) = ~~equal to the~~ (select~~ion of the~~ count(*) ~~of all rows~~ from ~~the~~ musical preferences ~~table~~ where ~~the~~ Musical_Preferences. customer ID ~~in the musical preferences table matches the~~ = Customers.customer ID ~~in the customers table~~)
（依次基于顾客 ID 相同将顾客表内连接到音乐风格喜好表、基于风格 ID 相同内连接到演唱组合风格表、基于演唱组合 ID 相同内连接到演唱组合表；按顾客的 ID、名和姓以及演唱组合的 ID 和艺名分组；对于每个分组，计算其中风格 ID 列不为 Null 的行数，再计算音乐风格喜好表中顾客 ID 与该分组的顾客 ID 相同的行数，如果它们相等，就从该分组中选择顾客 ID、顾客名和顾客姓、演唱组合 ID、演唱组合艺名以及风格 ID 不为 Null 的行数）

SQL

```
SELECT Customers.CustomerID,
  Customers.CustFirstName,
  Customers.CustLastName,
  Entertainers.EntertainerID,
  Entertainers.EntStageName,
  Count(Musical_Preferences.StyleID) AS
   CountOfStyleID
FROM ((Customers INNER JOIN Musical_Preferences
  ON Customers.CustomerID =
     Musical_Preferences.CustomerID)
INNER JOIN Entertainer_Styles
  ON Musical_Preferences.StyleID =
     Entertainer_Styles.StyleID)
```

```
            INNER JOIN Entertainers
            ON Entertainers.EntertainerID =
               Entertainer_Styles.EntertainerID
            GROUP BY Customers.CustomerID,
               Customers.CustFirstName,
               Customers.CustLastName,
               Entertainers.EntertainerID,
               Entertainers.EntStageName
            HAVING Count(Musical_Preferences.StyleID) =
              (SELECT Count(*)
               FROM Musical_Preferences
               WHERE Musical_Preferences.CustomerID =
                     Customers.CustomerID);
```

❖ **说明** 使用下面的 SQL 代码语句可获得同样的结果：

```
HAVING Count(Musical_Preferences.StyleID) =
  (SELECT Count(*)
  FROM Entertainer_Styles
  WHERE Entertainer_Styles.EntertainerID =
        Entertainers.EntertainerID)
```

CH18_Entertainers_Fully_Match_Customers_Style（8 行）

CustomerID	CustFirstName	CustLastName	EntertainerID	EntStageName	CountOfStyleID
10002	Deb	Waldal	1003	JV & the Deep Six	2
10003	Peter	Brehm	1002	Topazz	2
10005	Elizabeth	Hallmark	1009	Katherine Ehrlich	2
10005	Elizabeth	Hallmark	1011	Julia Schnebly	2
10008	Darren	Gehring	1001	Carol Peacock Trio	2
10010	Zachary	Ehrlich	1005	Jazz Persuasion	3
10012	Kerry	Patterson	1001	Carol Peacock Trio	2
10013	Estella	Pundt	1005	Jazz Persuasion	2

这之所以可行，是因为顾客喜欢的每种音乐风格以及演唱组合演奏的每种音乐风格在数据库中都只出现一次。请注意，你无须知道要匹配多少种风格——这项工作查询会替你完成。我在结果中包含 CountOfStyleID 列，只是为了证明喜欢的音乐风格数因顾客而异。试想，如果上述列表中的顾客致电要求推荐演唱组合，这将是一个非常好的销售工具。此时经纪人可自信满满地至少推荐一个这样的演唱组合：顾客喜欢的音乐风格都是他们演奏的音乐风格。

18.4 语句举例

至此，你知道了如何编写查询来解决否定型和多条件型问题，还见过了一些你能够解决的问题类型。下面来看一些示例，它们演示了如何解决各种否定型和多条件型问题。这些示例都来自示例数据库，演示了如何使用连接、IN、EXISTS 和分组来解决需要使用多个查找条件的问题。

❖ **说明** 本书前言提醒过，在你使用的数据库系统中，行的排列顺序不一定与本书示例中显示的相同——除非使用 ORDER BY 子句。即便指定了排序方式，在数据库系统返回的结果中，对于 ORDER BY 子句中未包含的列，它们的排列顺序也可能不同，这是因为优化方法因数据库系统而异。

如果你在 Microsoft SQL Server 中运行示例，则从视图中选择行时，将不会遵循其中的 ORDER BY 子句指定的排列顺序。要遵循 ORDER BY 子句指定的排列顺序，必须打开视图的设计，并在显示界面中执行它。

另外，当你使用 GROUP BY 子句时，数据库系统返回的结果集看起来像是根据该子句中指定的列进行了排序。这是因为有些优化器首先会在内部对数据排序，以提高 GROUP BY 的处理速度。别忘了，如果要按特定的顺序排列，必须使用 ORDER BY 子句。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库中，名称以 CH18 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明**　别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录 B。这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。为简化查询编写过程，我在下面的示例中将转换和整理合二为一了。

- **Sales Orders 数据库**

“找出所有订购了自行车和头盔的顾客。”

> ❖ **说明**　第 8 章解决过这个问题，方法是将两个 SELECT DISTINCT 子查询连接起来，但这里使用 EXISTS 来解决。

转换/整理

Select customer ID, cust~~omer~~ first name, ~~and~~ cust~~omer~~ last name from ~~the~~ customers ~~table~~ where ~~there~~ exists ~~some row in~~ (SELECT * FROM ~~the~~ orders ~~table~~ inner join~~ed with the~~ order details ~~table~~ on orders.order number ~~in the orders table equals~~ = order_details.order number ~~in the order details table, and then~~ inner join~~ed with the~~ products ~~table~~ on products.product ID ~~in the products table equals~~ = order_details.product ID ~~in the order details table~~ where product name ~~contains~~ LIKE ‘%Bike’ and Orders.customer ID ~~in the orders table equals~~ = ~~the~~ Customers.customer ID ~~in the customers table),~~ and ~~there also~~ exists ~~some row in~~ (SELECT * FROM ~~the~~ orders ~~table~~ inner join~~ed with the~~ order details ~~table~~ on orders.order ID ~~in the orders table equals~~ = order_details.order ID ~~in the order details table, and then~~ inner join~~ed with the~~ products ~~table~~ on products.product ID ~~in the products table equals~~ = order_details.product ~~ID in the order details table~~ where product name ~~contains~~ LIKE ‘%Helmet’ and ~~the~~ Orders.customer ID ~~in the orders table equals~~ = ~~the~~ Customers.customer ID ~~in the customers table~~)（按如下方式处理顾客表中的每行：依次基于订单号相同将订单表内连接到订单详情表、基于商品 ID 相同内连接到商品表，选择商品名包含 Bike 且顾客 ID 与该行中顾客 ID 相同的行的行；依次基于订单号相同将订单表内连接到订单详情表、基于商品 ID 相同内连接到商品表，选择商品名包含 Helmet 且顾客 ID 与该行中顾客 ID 相同的行；如果前面两个结果集都不是空的，就从该行中选择顾客的 ID、名和姓）

SQL

```
SELECT Customers.CustomerID,
   Customers.CustFirstName,
   Customers.CustLastName
FROM Customers
WHERE EXISTS
   (SELECT *
    FROM (Orders INNER JOIN Order_Details
      ON Orders.OrderNumber =
         Order_Details.OrderNumber)
   INNER JOIN Products
      ON Products.ProductNumber =
         Order_Details.ProductNumber
    WHERE Products.ProductName LIKE '%Bike'
      AND Orders.CustomerID =
          Customers.CustomerID)
AND EXISTS
   (SELECT *
    FROM (Orders INNER JOIN Order_Details
      ON Orders.OrderNumber =
         Order_Details.OrderNumber)
    INNER JOIN Products
      ON Products.ProductNumber =
         Order_Details.ProductNumber
    WHERE Products.ProductName LIKE '%Helmet'
      AND Orders.CustomerID =
          Customers.CustomerID)
```

CH18_Cust_Bikes_And_Helmets_EXISTS（21 行）

CustomerID	CustFirstName	CustLastName
1002	William	Thompson
1004	Robert	Brown
1005	Dean	McCrae

（续）

CustomerID	CustFirstName	CustLastName
1006	John	Viescas
1007	Mariya	Sergienko
1008	Neil	Patterson
1009	Andrew	Cencini
1010	Angel	Kennedy
1012	Liz	Keyser
1013	Rachel	Patterson
	<< 其他行 >>	

“找出所有未订购自行车和轮胎的顾客。”

❖ **说明** 这里稍微简化了点儿，因为我知道自行车的类别 ID 为 2，而轮胎的类别 ID 为 6。如果不知道这些，就得再连接到 Categories 表并在其中查找 Bikes 和 Tires。

转换/整理

Select customer ID, cust~~omer~~ first name, ~~and~~ cust~~omer~~ last name from ~~the~~ customers ~~table~~ where customer ID ~~is~~ not in ~~the~~ (selec~~tion of~~ customer ID from ~~the~~ orders ~~table~~ inner join~~ed with the~~ order details ~~table~~ on Orders.order number ~~in the orders table equals~~ = Order_Details.order number ~~in the order details table, and then~~ inner join~~ed with the~~ products ~~table~~ on Products.product ID ~~in the products table equals~~ = Order_Details.product ID ~~in the order details table~~ where product category ~~is~~ = 2)~~,~~ and customer ID ~~is~~ not in ~~the~~ (selec~~tion of~~ customer ID from ~~the~~ orders ~~table~~ inner join~~ed with the~~ order details ~~table~~ on Orders.order number ~~in the orders table equals~~ = Order_Details.order number ~~in the order details table, and then~~ inner join~~ed with the~~ products ~~table~~ on Products.product ID ~~in the products table equals~~ = Order_Details.product ID ~~in the order details table~~ where product category ID ~~is~~ = 6)（依次基于订单号相同将订单表内连接到订单详情表、基于商品 ID 相同内连接到商品表，从商品类别 ID 为 2 的行中选择顾客 ID，生成第一个列表；依次基于订单号相同将订单表内连接到订单详情表、基于商品 ID 相同内连接到商品表，从商品类别 ID 为 6 的行中选择顾客 ID，生成第二个列表；在顾客表中，从顾客 ID 不在上述两个列表中的行中选择顾客的 ID、名和姓）

SQL

```
SELECT Customers.CustomerID, Customers.
CustFirstName,
  Customers.CustLastName
FROM Customers
WHERE Customers.CustomerID NOT IN
   (SELECT CustomerID
    FROM (Orders INNER JOIN Order_Details
      ON Orders.OrderNumber =
         Order_Details.OrderNumber)
    INNER JOIN Products
      ON Order_Details.ProductNumber =
         Products.ProductNumber
    WHERE Products.CategoryID = 2)
AND Customers.CustomerID NOT IN
 (SELECT CustomerID
  FROM (Orders INNER JOIN Order_Details
    ON Orders.OrderNumber =
       Order_Details.OrderNumber)
  INNER JOIN Products
    ON Order_Details.ProductNumber =
       Products.ProductNumber
  WHERE Products.CategoryID = 6)
```

CH18_Customers_Not_Bikes_Or_Tires_NOTIN_2（2 行）

CustomerID	CustFirstName	CustLastName
1022	Caleb	Viescas
1028	Jeffrey	Tirekicker

❖ **说明** 在任何查找没有购买某种商品的查询的结果集中，都包含 Jeffrey Tirekicker，因为这位顾客什么都没买。为了验证这一点，请参阅 CH18_Customers_No_Orders_JOIN 和 CH18_Customers_No_Orders_NOT_IN。

- **Entertainment Agency 数据库**

“列出给顾客 Berg 和 Hallmark 演出过的演唱组合。”

❖ **说明** 第 8 章解决过这个问题，方法是连接两个复杂的表查询，但这里将使用 EXISTS。

转换/整理

Select entertainer ID, ~~and~~ ent~~ertainer~~ stage name from ~~the~~ entertainers ~~table~~ where ~~there~~ exists (SELECT * ~~some row~~ from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on customers.customer ID ~~in the customers table matches~~ = engagements.customer ID ~~in the engagements table~~ where customer last name ~~is~~ = 'Berg' and ~~the~~ engagements ~~table~~ entertainer ID ~~equals~~ = ~~the~~ Entertainers.entertainer ID ~~in the entertainers table),~~ and ~~there also~~ exists (SELECT * ~~some row~~ from ~~the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on customers.customer ID ~~in the customers table matches~~= engagements.customer ID ~~in the engagements table~~ where cust~~omer~~ last name ~~is~~ = 'Hallmark' and ~~the~~ Engagements.entertainer ID ~~in the engagements table equals~~ = ~~the~~ Entertainers.entertainer ID ~~in the entertainers table~~)（在演唱组合表中，对每行都做如下处理：基于顾客 ID 相同将顾客表内连接到演出合约表，选择顾客姓 Berg 且演唱组合 ID 与该行中演唱组合相同的行，生成第一个结果集；基于顾客 ID 相同将顾客表内连接到演出合约表，选择顾客姓 Hallmark 且演唱组合 ID 与该行中演唱组合相同的行，生成第二个结果集；如果这两个结果集都不是空的，就从该行中选择演唱组合的 ID 和艺名）

SQL

```
SELECT Entertainers.EntertainerID,
   Entertainers.EntStageName
FROM Entertainers
WHERE EXISTS
   (SELECT *
    FROM Customers INNER JOIN Engagements
      ON Customers.CustomerID =
         Engagements.CustomerID
    WHERE Customers.CustLastName = 'Berg'
      AND Engagements.EntertainerID =
          Entertainers.EntertainerID)
AND EXISTS
   (SELECT *
    FROM Customers INNER JOIN Engagements
      ON Customers.CustomerID =
         Engagements.CustomerID
    WHERE Customers.CustLastName = 'Hallmark'
      AND Engagements.EntertainerID =
          Entertainers.EntertainerID)
```

CH18_Entertainers_Berg_AND_Hallmark_EXISTS（4 行）

EntertainerID	EntStageName
1001	Carol Peacock Trio
1003	JV & the Deep Six
1006	Modern Dance
1008	Country Feeling

“选择从未将演唱风格包含 Country 或 Country Rock 的演唱组合推销出去的经纪人。”

转换/整理

Select agent ID, ag~~ent~~ first name, ~~and~~ ag~~ent~~ last name from ~~the~~ agents ~~table~~ where agent ID ~~is~~ not in ~~the~~ (select~~ion of~~ agent ID from ~~the~~ engagements ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on Engagements.entertainer ID ~~in the engagements table equals~~ = Entertainers.entertainer ID ~~in the entertainers table, and then~~ inner join~~ed with the~~ entertainer styles ~~table~~ on Entertainers.entertainer ID ~~in the entertainers table equals~~ = Entertainer_ Styles.entertainer ID ~~in the entertainer styles table, and then~~ inner join~~ed with the~~ musical styles ~~table~~ on Entertainer_Styles.style ID ~~in the entertainer styles table equals~~ = Musical_Styles.style ID ~~in the musical styles table~~ where style name ~~is~~ in ('Country', ~~or~~ 'Country Rock'))（依次基于经纪人 ID 相同将演唱组合表内连接到演出合约表、基于演唱组合 ID 相同内连接到演出合约表、基于演唱组合 ID 相同内连接到演唱组合风格表、基于风格 ID 相同连接到音乐风格表，从风格名为 Country 或 Country Rock 的行中选择经纪人 ID，生成一个列表；在经纪人表中，从经纪人 ID 不在上述列表中的行中选择经纪人的 ID、名和姓）

SQL

```
SELECT Agents.AgentID, Agents.AgtFirstName,
   Agents.AgtLastName
FROM Agents
WHERE Agents.AgentID NOT IN
   (SELECT Engagements.AgentID
    FROM ((Engagements INNER JOIN Entertainers
```

```
      ON Engagements.EntertainerID =
         Entertainers.EntertainerID)
    INNER JOIN Entertainer_Styles
      ON Entertainers.EntertainerID =
        Entertainer_Styles.EntertainerID)
    INNER JOIN Musical_Styles
      ON Entertainer_Styles.StyleID =
         Musical_Styles.StyleID
    WHERE Musical_Styles.StyleName IN
      ('Country', 'Country Rock'));
```

CH18_Agents_Not_Book_Country_CountryRock（3 行）

AgentID	AgtFirstName	AgtLastName
2	Scott	Bishop
8	Maria	Patterson
9	Daffy	Dumbwit

❖ **说明** 在任何查找未将某个演唱组合推销出去的经纪人的查询的结果中，都包含 Daffy Dumbwit，因为他未将任何演唱组合推销出去。

- **School Scheduling 数据库**

"列出艺术课和计算机课的成绩都不低于 85 的学生。"

❖ **说明** 第 8 章演示了如何通过连接两个 DISTINCT 子查询来解决这个问题，这里将演示如何使用 IN 来解决。

转换/整理

Select student ID, stud~~ent~~ first name, ~~and~~ stud~~ent~~ last name from ~~the~~ students ~~table~~ where student ID ~~is~~ in ~~the~~ (selecti~~on of~~ student ID from ~~the~~ student schedules ~~table~~ inner join~~ed with the~~ classes ~~table~~ on Classes.student ID ~~in the classes table equals~~ = Student_Schedules.student ID ~~in the student schedules table, then~~ inner join~~ed with the~~ subjects ~~table~~ on Subjects.subject ID ~~in the subjects table equals~~ = Classes.subject ID ~~in the classes styles table, and then~~ inner join~~ed with the~~ categories ~~table~~ on Categories.category ID ~~in the categories table equals~~ = Subjects.category ID ~~in the subjects table~~ where category description ~~is equal to~~ = 'art' and grade ~~is greater than or equal to~~ >= 85) and student ID ~~is~~ in ~~the~~ (selecti~~on of~~ student ID from ~~the~~ student schedules ~~table~~ inner join~~ed with the~~ classes ~~table~~ on Classes.student ID ~~in the classes table equals~~ = Student_Schedules.student ID ~~in the student schedules table, then~~ inner join~~ed with the~~ subjects ~~table~~ on Subjects.subject ID ~~in the subjects table equals~~ = Classes.subject ID ~~in the classes styles table, and then~~ inner join~~ed with the~~ categories ~~table~~ on Categories.category ID ~~in the categories table equals~~ = Subjects.category ID ~~in the subjects table~~ where category description ~~contains~~ LIKE '%computer%' and grade ~~is greater than or equal to~~ >= 85)（依次基于学生 ID 相同将学生选课表内连接到课程表、基于科目 ID 相同内连接到科目表、基于类别 ID 相同内连接到类别表，从类别描述为 Art 且成绩不低于 85 的行中选择学生 ID，生成第一个列表；依次基于学生 ID 相同将学生选课表内连接到课程表、基于科目 ID 相同内连接到科目表、基于类别 ID 相同内连接到类别表，从类别描述包含 computer 且成绩不低于 85 的行中选择学生 ID，生成第二个列表；在学生表中，从学生 ID 出现在了上述两个列表中的行中选择学生的 ID、名和姓）

SQL

```
SELECT Students.StudentID,
    Students.StudFirstName, Students.StudLastName
FROM Students
WHERE Students.StudentID IN
   (SELECT Student_Schedules.StudentID
    FROM ((Student_Schedules INNER JOIN Classes
      ON Classes.ClassID =
         Student_Schedules.ClassID)
    INNER JOIN Subjects
      ON Subjects.SubjectID = Classes.SubjectID)
    INNER JOIN Categories
      ON Categories.CategoryID = Subjects.
         CategoryID
    WHERE Categories.CategoryDescription = 'Art'
      AND Student_Schedules.Grade >= 85)
AND Students.StudentID IN
   (SELECT Student_Schedules.StudentID
    FROM ((Student_Schedules INNER JOIN Classes
      ON Classes.ClassID =
         Student_Schedules.ClassID)
```

```
        INNER JOIN Subjects
          ON Subjects.SubjectID = Classes.SubjectID)
        INNER JOIN Categories
          ON Categories.CategoryID = Subjects.
             CategoryID
        WHERE Categories.CategoryDescription LIKE
              '%Computer%'
          AND Student_Schedules.Grade >= 85);
```

CH18_Good_Art_CS_Students_IN（1 行）

StudentID	StudFirstName	StudLastName
1011	John	Kennedy

“显示这样的学生，即他注册了某门课程，却没有完成该课程的先修课程。”

❖ **说明**　这是一个有趣的否定型多条件问题。注册的课程数因学生而异，未注册或完成的先修课程也因学生而异，而你必须在查询中对每位学生的这两种课程进行比较（这里假设允许同时注册某门课程及其先修课程）。

我们换种说法，以便让你更清楚地知道该如何解决这个问题。

“显示这样的学生，即他注册了某门课程，却没有完成该课程的先修课程。所谓完成了先修课程，指的是先修课程的注册时间不能晚于当前课程，且学生没有将其退掉。”

转换/整理

Select student ID, stud~~ent~~ first name, stud~~ent~~ last name start date, subject code, subject name, ~~and~~ subject prereq from ~~the~~ students ~~table~~ inner join~~ed with the~~ student schedules ~~table~~ on Students.student ID ~~in the students table equals~~ = Student_Schedules.student ID ~~in the student schedules table, then~~ inner join~~ed with the~~ classes ~~table~~ on Classes.class ID ~~in the classes table equals~~ = Student_Schedules.class ID ~~in the student schedules table, and then~~ inner joined ~~with the~~ subjects ~~table~~ on Subjects.subject ID ~~in the subjects table equals~~ = Classes.subject ID ~~in the classes table~~ where subject prereq is not null and subject prereq ~~is~~ not in ~~the~~ (selec~~tion of~~ subject code from ~~the~~ subjects ~~table~~ inner join~~ed with the~~ classes ~~table aliased~~ as c2 on Subjects.subject ID ~~in the subjects table equals~~ = C2.subject ID ~~in the c2 aliased table, and then~~ inner join~~ed with the~~ student schedules ~~table~~ on C2.class ID ~~in the c2 aliased table equals~~ = Student_Schedules.class ID ~~in the student schedules table, and then~~ inner join~~ed with the~~ student class status ~~table~~ on Student_Schedules.class status ~~in the student schedules table equals~~ = Student_Class_Status.class status ~~in the student class status table~~ where class status description ~~does not equal~~ <> 'withdrew' and Student_Schedules.student ID ~~in the student schedules table equals~~ = Students.student ID ~~in the students table~~ and C2.start date ~~in the aliased c2 table is less than or equal to~~ <= Classes.start date ~~in the classes table~~)［依次基于学生 ID 相同将学生表内连接到学生选课表、基于课程 ID 相同内连接到课程表、基于科目 ID 相同内连接到科目表。对于每个先修课程不为 Null 的行，做如下处理：依次基于科目 ID 相同将科目表内连接到课程表（别名为 C2）、基于课程 ID 相同内连接到学生选课表、基于课程状态相同内连接到学生课程状态表；从课程状态描述不为 withdrew、学生 ID 与当前行学生 ID 相同且开始日期不晚于当前行开始日期的行中选择科目代码，生成一个列表；如果当前行的先修课程不在这个列表中，就从当前行选择学生的 ID、名和姓以及开始日期、科目代码、科目名称和先修课程］

SQL

```
SELECT Students.StudentID,
   Students.StudFirstName,
   Students.StudLastName, Classes.StartDate,
   Subjects.SubjectCode, Subjects.SubjectName,
   Subjects.SubjectPreReq
FROM ((Students INNER JOIN Student_Schedules
  ON Students.StudentID =
     Student_Schedules.StudentID)
INNER JOIN Classes
  ON Classes.ClassID =
     Student_Schedules.ClassID)
INNER JOIN Subjects
  ON Subjects.SubjectID = Classes.SubjectID
WHERE Subjects.SubjectPreReq IS NOT NULL
  AND Subjects.SubjectPreReq NOT IN
   (SELECT Subjects.SubjectCode
FROM ((Subjects INNER JOIN Classes AS C2
      ON Subjects.SubjectID = C2.SubjectID)
INNER JOIN Student_Schedules
     ON C2.ClassID = Student_Schedules.ClassID)
    INNER JOIN Student_Class_Status
      ON Student_Schedules.ClassStatus =
```

```
        Student_Class_Status.ClassStatus
WHERE
  Student_Class_Status.ClassStatusDescription
    <> 'Withdrew'
   AND Student_Schedules.StudentID =
      Students.StudentID
   AND C2.StartDate <= Classes.StartDate);
```

CH18_Students_Missing_Prerequisites（5 行）

StudentID	StudFirstName	StudLastName	StartDate	SubjectCode	SubjectName	Prerequisite
1005	Doris	Hartwig	2017-09-11	ENG	Composition-Intermediate	ENG 101
1007	Elizabeth	Hallmark	2017-09-11	ENG	Composition-Intermediate	ENG 101
1012	Sarah	Thompson	2017-09-11	ENG	Composition-Intermediate	ENG 101
1014	Kendra	Bonnicksen	2017-09-11	ENG	Composition-Intermediate	ENG 101
1018	Richard	Lum	2017-09-11	ENG	Composition-Intermediate	ENG 101

- **Bowling League 数据库**

“列出在保龄球馆 Thunderbird Lanes、Totem Lanes 和 Bolero Lanes 举行的比赛中，都以不超过 190 的让分得分获胜过的投球手，以及比赛的场次、局次、让分得分、联赛日期和联赛地点。”

❖ **说明** 首先需要找出至少在这三个保龄球馆之一进行的比赛中，以不超过 190 的让分得分获胜过的投球手，再检查各个投球手是否在这三个保龄球馆进行的比赛中都这样过（别忘了，不是 IN (a, b, c)，而是 IN (a) AND IN (b), AND in (c)）。

转换/整理

Select bowler ID, bowler first name, bowler last name, match ID, game number, handicap score, tourney date, ~~and~~ tourney location from ~~the~~ bowlers ~~table~~ inner join~~ed with the~~ bowler scores ~~table~~ on Bowlers.bowler ID ~~in the bowlers table equals~~ = Bowler_Scores.bowler ID ~~in the bowler scores table, then~~ inner join~~ed with the~~ tourney matches ~~table~~ on Bowler_Scores.match ID ~~in the bowler scores table equals~~ = Tourney_Matches.match ID ~~in the tourney matches table, and then~~ inner join~~ed with the~~ tournaments ~~table~~ on Tournaments.tourney ID ~~in the tournaments table equals~~ = Tourney_Matches.tourney ID ~~in the tourney matches table~~ where handicap score ~~is less than or equal to~~ <= 190 and won game ~~equals~~ = 1 and tourney location ~~is~~ in ~~the list~~ (‘Thunderbird Lanes’, ‘Totem Lanes’, and ‘Bolero Lanes’) and bowler ID ~~is~~ in ~~the~~ (selec~~tion of~~ bowler ID from ~~the~~ tournaments ~~table~~ inner join~~ed with the~~ tourney matches ~~table~~ on Tournaments.tourney ID ~~in the tournaments table equals~~ = Tourney_Matches.tourney ID ~~in the tourney matches table, and then~~ inner join~~ed with the~~ bowler scores ~~table~~ on Tourney_Matches.match ID ~~in the tourney matches table equals~~ = Bowler_Scores.match ID ~~in the bowler scores table~~ where won game ~~equals~~ = 1 and handicap score ~~is less than or equal to~~ <= 190 and tourney location ~~equals~~ = ‘Thunderbird Lanes’) and bowler ID ~~is~~ in ~~the~~ (selec~~tion of~~ bowler ID from ~~the~~ tournaments ~~table~~ inner join~~ed with the~~ tourney matches ~~table~~ on Tournaments.tourney ID ~~in the tournaments table equals~~ = Tourney_Matches.tourney ID ~~in the tourney matches table, and then~~ inner join~~ed with the~~ bowler scores ~~table~~ on Tourney_Matches.match ID ~~in the tourney matches table equals~~ = Bowler_Scores.match ID ~~in the bowler scores table~~ where won game ~~equals~~ = 1 and handicap score ~~is less than or equal to~~ <= 190 and tourney location ~~equals~~ = ‘Totem Lanes’) and bowler ID ~~is~~ in ~~the~~ (selec~~tion of~~ bowler ID from ~~the~~ tournaments ~~table~~ inner join~~ed with the~~ tourney matches ~~table~~ on Tournaments.tourney ID ~~in the tournaments table equals~~ = Tourney_Matches.tourney ID ~~in the tourney matches table, and then~~ inner join~~ed with the~~ bowler scores ~~table~~ on Tourney_Matches.match ID ~~in the tourney matches table equals~~ = Bowler_Scores.match ID ~~in the bowler scores table~~ where won game ~~equals~~ = 1 and handicap score ~~is less than or equal to~~ <= 190 and tourney location ~~equals~~ = ‘Bolero Lanes’)（依次基于投球手 ID 将投球手表内连接到投球手得分表、基于场次 ID 相同内连接到联赛场次表、基于联赛 ID 相同内连接到联赛表。对于让分得分不超过 190、是否获胜为 1 且联赛地点为 Thunderbird Lanes、Totem Lanes 或 Bolero Lanes 的每一行，做如下处理：依次基于联赛 ID 相同将联赛表内连接到联赛场次表、基于场次 ID 相同内连接到投球手得分表，从是否获胜为 1、让分得分不超过 190 且联赛地点为 Thunderbird Lanes 的行中，选择投球手 ID，生成第一个列表；依次基于联赛 ID 相同将联赛表内连接到联赛场次表、基于场次 ID 相同内连接到投球手得分表，从是否获胜为 1、让分得分不超过 190 且联赛地点为 Totem Lanes 的行中，选择投球手 ID，生成第二个列表；依次基于联赛 ID 相同将联赛表内连接到联赛场次表、基于场次 ID 相同内连接到投球手得分表，从是否获胜为 1、让分得分不超过 190 且联赛地点为 Bolero Lanes 的行中，选择投球手 ID，生成第三个列表；如果当前行的投球手 ID 包含在上述全部三个列表中，就从当前行选择投球手的 ID、名和姓以及场次 ID、局次、让分得分、联赛日期和联赛地点）

SQL

```
SELECT Bowlers.BowlerID, Bowlers.BowlerFirstName,
  Bowlers.BowlerLastName, Bowler_Scores.MatchID,
  Bowler_Scores.GameNumber,
  Bowler_Scores.HandiCapScore,
  Tournaments.TourneyDate,
  Tournaments.TourneyLocation
FROM ((Bowlers INNER JOIN Bowler_Scores
  ON Bowlers.BowlerID = Bowler_Scores.BowlerID)
INNER JOIN Tourney_Matches
  ON Bowler_Scores.MatchID =
     Tourney_Matches.MatchID)
INNER JOIN Tournaments
  ON Tournaments.TourneyID =
     Tourney_Matches.TourneyID
WHERE (Bowler_Scores.HandiCapScore <= 190)
  AND (Bowler_Scores.WonGame = 1
  AND (Tournaments.TourneyLocation IN
    ('Thunderbird Lanes', 'Totem Lanes',
       'Bolero Lanes'))
  AND (Bowlers.BowlerID IN
   (SELECT Bowler_Scores.BowlerID
    FROM (Tournaments INNER JOIN Tourney_Matches
      ON Tournaments.TourneyID =
         Tourney_Matches.TourneyID)
    INNER JOIN Bowler_Scores
      ON Tourney_Matches.MatchID =
         Bowler_Scores.MatchID
    WHERE Bowler_Scores.WonGame = 1
      AND Bowler_Scores.HandiCapScore <=190
      AND Tournaments.TourneyLocation =
        'Thunderbird Lanes'))
  AND (Bowlers.BowlerID IN
   (SELECT Bowler_Scores.BowlerID
    FROM (Tournaments INNER JOIN Tourney_Matches
      ON Tournaments.TourneyID =
         Tourney_Matches.TourneyID)
    INNER JOIN Bowler_Scores
      ON Tourney_Matches.MatchID =
         Bowler_Scores.MatchID
    WHERE Bowler_Scores.WonGame = 1
      AND Bowler_Scores.HandiCapScore <=190
      AND Tournaments.TourneyLocation =
        'Totem Lanes'))
  AND (Bowlers.BowlerID IN
   (SELECT Bowler_Scores.BowlerID
    FROM (Tournaments INNER JOIN Tourney_Matches
      ON Tournaments.TourneyID =
         Tourney_Matches.TourneyID)
    INNER JOIN Bowler_Scores
      ON Tourney_Matches.MatchID =
         Bowler_Scores.MatchID
    WHERE Bowler_Scores.WonGame = 1
      AND Bowler_Scores.HandiCapScore <=190
      AND Tournaments.TourneyLocation =
        'Bolero Lanes'));
```

CH18_Bowlers_Won_LowScore_TBird_Totem_Bolero_RIGHT（11 行）

Bowler ID	Bowler FirstName	Bowler LastName	Match ID	Game Number	HandiCap Score	Tourney Date	Tourney Location
13	Elizabeth	Hallmark	10	1	189	2017-09-18	Bolero Lanes
13	Elizabeth	Hallmark	24	3	190	2017-10-09	Totem Lanes
13	Elizabeth	Hallmark	34	1	189	2017-10-30	Thunderbird Lanes
19	John	Viescas	7	3	185	2017-09-11	Thunderbird Lanes
19	John	Viescas	12	1	181	2017-09-18	Bolero Lanes
19	John	Viescas	36	1	179	2017-10-30	Thunderbird Lanes
19	John	Viescas	52	2	185	2017-11-27	Totem Lanes
25	Megan	Patterson	7	1	188	2017-09-11	Thunderbird Lanes
25	Megan	Patterson	21	1	189	2017-10-09	Totem Lanes
25	Megan	Patterson	35	1	187	2017-10-30	Thunderbird Lanes
25	Megan	Patterson	39	2	181	2017-11-06	Bolero Lanes

> ❖ **说明** 示例数据库 Bowling League 提供了使用 IN 的错误解决方案，名为 CH18_Bowlers_Won_LowScore_TBird_Totem_Bolero_WRONG。

“列出在 Thunderbird Lanes 和 Bolero Lanes 进行的比赛中，原始得分从未超过 165 的投球手。”

转换/整理

Select bowler ID, bowler last name, ~~and~~ bowler first name from ~~the~~ bowlers ~~table~~ where bowler ID ~~is~~ not in ~~the~~ (selecti~~on of~~ bowler ID from ~~the~~ tournaments ~~table~~ inner join~~ed with the~~ tourney matches ~~table~~ on Tournaments.tourney ID ~~in the tournaments table equals~~ = Tourney_Matches.tourney ID ~~in the tourney matches table, then~~ inner join~~ed with the~~ bowler scores ~~table~~ on Tourney_Matches.match ID ~~in the tourney matches table equals~~ = Bowler_Scores.match ID ~~in the bowler scores table~~ where raw score ~~is greater than~~ > 165 and tourney location ~~is~~ in ~~the list of~~ (‘Thunderbird Lanes’, ~~and~~ ‘Bolero Lanes’))（依次基于联赛 ID 相同将联赛表内连接到联赛场次表、基于场次 ID 相同内连接到投球手得分表，从原始得分超过 165 且联赛地点为 Thunderbird Lanes 或 Bolero Lanes 的行中，选择投球手 ID，生成一个列表；在投球手表中，从投球手 ID 不在上述列表中的行中，选择投球手的 ID、姓和名）

SQL

```
SELECT Bowlers.BowlerID, Bowlers.
   BowlerLastName, Bowlers.BowlerFirstName
FROM Bowlers
WHERE Bowlers.BowlerID NOT IN
(SELECT Bowler_Scores.BowlerID
FROM (Tournaments INNER JOIN Tourney_Matches
      ON Tournaments.TourneyID =
         Tourney_Matches.TourneyID)
    INNER JOIN Bowler_Scores
      ON Tourney_Matches.MatchID =
         Bowler_Scores.MatchID
WHERE (Bowler_Scores.RawScore > 165)
  AND (Tournaments.TourneyLocation IN
  ('Thunderbird Lanes', 'Bolero Lanes')))
```

CH18_Bowlers_LTE_165_Thunderbird_Bolero（15 行）

BowlerID	BowlerLastName	BowlerFirstName
1	Fournier	Barbara
4	Sheskey	Sara
5	Patterson	Ann
8	Viescas	Stephanie
9	Black	Alastair
12	Viescas	Carol
13	Hallmark	Elizabeth
16	Sheskey	Richard
17	Hernandez	Kendra
20	Viescas	Suzanne
	<< 其他行 >>	

- **Recipes 数据库**

“显示菜品 Irish Stew、Pollo Picoso 和 Roast Beef 都未使用的食材。”

转换/整理

Select ingredient ID, ~~and~~ ingredient name from ~~the~~ ingredients ~~table~~ where ingredient ID ~~is~~ not in ~~the~~ (selecti~~on of~~ ingredient ID from ~~the~~ recipe ingredients ~~table~~ inner join~~ed with the~~ recipes ~~table~~ on Recipe_Ingredients.recipe ID ~~in the recipe ingredients table equals~~ = Recipes.recipe ID ~~in the recipes table~~ where recipe title is in ~~the list of~~ (‘Irish Stew’, ‘Pollo Picoso’, ~~and~~ ‘Roast Beef’))（基于菜品 ID 相同将菜品食材表内连接到菜品表，从菜品名为 Irish Stew、Pollo Picoso 或 Roast Beef 的行中选择食材 ID，生成一个列表；在食材表中，从食材 ID 不在上述列表中的行中选择食材 ID 和食材名）

SQL

```
SELECT Ingredients.IngredientID,
  Ingredients.IngredientName
FROM Ingredients
WHERE Ingredients.IngredientID NOT IN
```

```
(SELECT Recipe_Ingredients.IngredientID
 FROM Recipe_Ingredients INNER JOIN Recipes
   ON Recipe_Ingredients.RecipeID =
      Recipes.RecipeID
 WHERE RecipeTitle IN
('Irish Stew', 'Pollo Picoso', 'Roast Beef'))
```

CH18_Ingredients_NOTIN_IrishStew_PolloPicoso_RoastBeef（67 行）

IngredientID	IngredientName
7	Tomato
8	Jalapeno
12	Halibut
13	Chicken, Fryer
14	Bacon
15	Romaine Lettuce
16	Iceberg Lettuce
17	Butterhead Lettuce
18	Scallop
19	Salmon
<< 其他行 >>	

“成对地列出使用的食材至少有 3 种相同的菜品。”

❖ **说明**　这与前面的一个问题类似，那个问题要求找出成对的顾客和演唱组合，其中演唱组合能够演奏顾客喜欢的所有音乐风格。你认为这里也需要使用菜品表和菜品食材表的两个副本吗？

转换/整理

Select Recipes.recipe ID ~~and~~ Recipes.recipe title ~~in the first copy of the recipes table and~~ R2.recipe ID AS R2ID ~~and~~ R2.recipe title AS R2Title ~~in the second copy of the recipes table, and the~~ count ~~of~~ (Recipe_Ingredients.recipe ID) AS CountOfRecipeID ~~in the first copy of the recipe ingredients table~~ from ~~the~~ recipes ~~table~~ inner join~~ed with the~~ recipe ingredients ~~table~~ on Recipes.recipe ID ~~in the recipes table equals~~ = Recipe_Ingredients.recipe ID ~~in the recipe ingredients table, then~~ inner join~~ed with the second copy of the~~ recipe ingredients ~~table~~ AS RI2 on Recipe_Ingredients.ingredient ID ~~in the recipe ingredients table equals~~ = RI2.ingredient ID ~~in the second copy of the recipe ingredients table, then~~ inner join~~ed with the second copy of the~~ recipes ~~table~~ AS R2 on R2.recipe ID ~~in the second copy of the recipes table equals~~ = RI2.recipe ID ~~in the second copy of the recipe ingredients table~~ where RI2. recipe ID ~~in the second copy of the recipe ingredients table is greater than~~ > Recipes.recipe ID ~~in the first copy of the recipes table,~~ group~~ed~~ by Recipes.recipe ID ~~in the first copy of the recipes table~~, Recipes.recipe title ~~in the first copy of the recipes table,~~ R2.recipe ID ~~in the second copy of the recipes table, and~~ R2.recipe title ~~in the second copy of the recipes table, and~~ having ~~the~~ count ~~of matching~~ (Recipe_Ingredients.ingredient ID) ~~in the recipe ingredients table greater than~~ > 3［依次基于菜品 ID 相同将菜品表内连接到菜品食材表、基于食材 ID 相同内连接到菜品食材表的第二个副本（别名为 RI2）、基于菜品 ID 相同内连接到菜品表的第二个副本（别名为 R2）；对于得到的结果集中的每行，对其中分别来自菜品食材表第二个副本和来自菜品表第一个副本的菜品 ID 进行比较，如果前者比后者大，就选择它；对于得到的结果集，按来自菜品表第一个副本的菜品 ID 和菜品名以及来自菜品表第二个副本的菜品 ID 和菜品名进行分组；对于每个分组，计算其行数（以来自菜品食材表第一个副本的菜品 ID 不为 Null 为准），如果大于 3，就从中选择来自菜品表第一个副本的菜品 ID 和菜品名、来自菜品名表第二个副本的菜品 ID 和菜品名（分别命名为 R2ID 和 R2Title）以及该分组包含的行数（以来自菜品食材表第一个副本的菜品 ID 不为 Null 为准，并给它指定别名 CountOfRecipeID）］

SQL

```
SELECT Recipes.RecipeID, Recipes.RecipeTitle,
  R2.RecipeID AS R2ID, R2.RecipeTitle AS R2Title,
  Count(Recipe_Ingredients.RecipeID)
     AS CountOfRecipeID
FROM ((Recipes INNER JOIN Recipe_Ingredients
  ON Recipes.RecipeID =
     Recipe_Ingredients.RecipeID)
INNER JOIN Recipe_Ingredients AS RI2
  ON Recipe_Ingredients.IngredientID =
     RI2.IngredientID)
```

```
    INNER JOIN Recipes AS R2
      ON R2.RecipeID = RI2.RecipeID
    WHERE RI2.RecipeID > Recipes.RecipeID
    GROUP BY Recipes.RecipeID, Recipes.RecipeTitle,
      R2.RecipeID, R2.RecipeTitle
    HAVING Count(Recipe_Ingredients.RecipeID) >= 3;
```

CH18_Recipes_AtLeast_3_Same_Ingredients（4 行）

RecipeID	RecipeTitle	R2ID	R2Title	CountOfRecipeID
2	Salsa Buena	11	Huachinango Veracruzana (Red Snapper, Veracruz style	3
2	Salsa Buena	13	Tourtière (French-Canadian Pork Pie)	3
6	Pollo Picoso	9	Roast Beef	3
11	Huachinango Veracruzana (Red Snapper, Veracruz style)		Tourtière (French-Canadian Pork Pie)	4

❖ **说明**　我通过检查确保来自菜品表第二个副本的菜品 ID 大于来自菜品表第一个副本的菜品 ID，以免将同一对菜品列出两次。按照 SQL 标准，可将这个筛选器放在连接两个 Recipe_Ingredients 表副本的 JOIN 子句中，但我选择将其放在了 WHERE 子句中，旨在确保与大多数数据库系统兼容。

18.5　小结

本章首先复习了集合，旨在帮助你理解如何解决涉及多个条件的否定型问题；还详细回顾了解决否定型多条件问题的 4 种方法，包括内连接、NOT IN、NOT EXISTS 和 GROUP BY/HAVING。

接下来介绍了 4 种解决多条件问题的方法：内连接、IN、EXISTS 和 GROUP BY/ HAVING。为了帮助你巩固概念，本章列举了 5 组针对示例数据库的示例，并确保对于每个示例数据库，都至少有一个针对它的多条件示例和一个否定型示例。

下一节列出了多个你可独自解决的问题。

18.6　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Order 数据库**

  (1) 显示未订购过自行车和轮胎的顾客。

  本章前面演示了如何使用 NOT IN 和多个子查询来解决这个问题。你能想出仅使用 NOT IN 的解决方案吗？解决方案见 CH18_Customers_Not_ Bikes_Or_Tires_NOTIN_1（2 行）。

  (2) 列出购买了自行车但没有购买头盔的顾客。

  首先，使用 EXISTS 和 NOT EXISTS 来解决这个问题，解决方案见 CH18_Cust_Bikes_No_Helmets_EXISTS（2 行）。作为加分练习，请使用 IN 和 NOT IN 来解决这个问题，解决方案见 CH18_Customer_Bikes_No_Helmets（2 行）。

  (3) 列出包含自行车但不包含头盔的订单。

  这看似与问题 2 相同，但其实并非如此。这里要求你列出的是订单，而不是顾客。请使用 EXISTS 和 NOT EXISTS 来解决这个问题，解决方案见 CH18_Orders_Bikes_No_Helmet_EXISTS（402 行）。

(4) 显示同时包含自行车和头盔的订单以及下单顾客。
请使用 EXISTS 来解决这个问题，解决方案见 CH18_Customers_Bikes_And_Helmets_Same_Order（184 行）。
(5) 显示提供配饰、车架或服装的供应商。
请使用 IN 来解决这个问题，解决方案见 CH18_Vendors_Accessories_CarRacks_Clothing（3 行）。

- **Entertainment Angency 数据库**

(1) 列出能够演奏爵士（Jazz）、节奏布鲁斯（Rhythm and Blues）或萨尔萨（Salsa）音乐风格的演唱组合。
使用 IN 来解决这个问题，但小心别大意失荆州，解决方案见 CH18_Entertainers_Jazz_RhythmBlues_Salsa_IN（1 行）。CH18_Entertainers_Jazz_RhythmBlues_Salsa_IN_WRONG 演示了使用 IN 的错误解决方案（4 行）。作为加分练习，请使用 GROUP BY 和 HAVING 来解决这个问题，解决方案见 CH18_Entertainers_Jazz_RhythmBlues_Salsa_HAVING（1 行）。
(2) 显示 2018 年 5 月 1 前的 90 天内都没有演出合约的演唱组合。
可使用 NOT IN 来解决这个问题，但请根据你使用的数据库系统小心选择合适的日期和时间函数，解决方案见 CH18_Entertainers_Not_Booked_90Days_Before_May1_2018（2 行）。
(3) 显示与 Topazz 和 Modern Dance 都没有签约的顾客。
可使用 NOT IN 以两种不同的方式解决这个问题，其中一种方式的解决方案见 CH18_Customers_Not_Booked_Topazz_Or_ModernDance（6 行）。
(4) 列出为 Hartwig、McCrae 和 Rosales 演出过的演唱组合。
解决方式有多种，其中使用 EXISTS 的解决方案见 CH18_Entertainers_Hartwig_McCrae_AND_ Rosales_EXISTS（2 行）。
(5) 显示未与任何演唱组合签约的顾客；显示没有签约过的演唱组合。
这两个问题都可使用简单的 NOT IN 来解决，解决方案见 CH18_Customers_No_Bookings_NOT_IN（2 行）和 CH18_Entertainers_Never_Booked_NOT_IN（1 行）。

- **School Scheduling 数据库**

(1) 显示艺术（Art）和计算机科学（Computer Science）的成绩都不低于 85 的学生。
本章前面演示了如何使用 IN 来解决这个问题。现在请你使用 EXISTS 来解决它，解决方案见 CH18_Good_Art_CS_Students_EXISTS（1 行）。
(2) 显示讲授了无资格讲授的课程的教员。
诀窍是在教员课程表中找出不在教员科目表中的行。
解决方案见 CH18_Staff_Teaching_NonAccredited_Classes（4 行）。
(3) 列出所有已结束课程的成绩都低于 80 的学生。
你可能猜到了，最佳的解决方法是使用 GROUP BY 和 HAVING，解决方案见 CH18_Students_Passed_All_Grade_GTE_80（3 行）。
(4) 下面是一些简单的否定型问题，请解决其中的 3 个：找出没有学生注册的课程；显示没有讲授任何课程的教员；显示从未退过课的学生；列出未入学的学生；找出没有分配任何教员的科目。
这些问题的解决方案见 CH18_Classes_No_Students_Enrolled_NOT_IN（118 行）、CH18_Staff_Not_Teaching_EXISTS（5 行）、CH18_Students_Never_Withdrawn_EXISTS（16 行）、CH18_Students_Not_Currently_Enrolled_NOT_IN（2 行）和 CH18_Subjects_No_Faculty_NOT_IN（1 行）。

- **Bowling League 数据库**

(1) 显示原始得分从未超过 150 的投球手。
一种解决方案见 CH18_Mediocre_Bowlers（7 行）。
(2) 显示在 Thunderbird Lanes 和 Bolero Lanes 举行的比赛中，原始得分都曾超过 170 的投球手。
我演示过如何通过内连接 SELECT DISTINCT 查询来解决这个问题，现在请你使用 EXISTS 来解决它，解决方案见 CH18_Good_Bowlers_TBird_And_Bolero_EXISTS（11 行）。

(3) 列出还未举行的联赛。

这个问题使用 NOT IN 很容易解决，解决方案见 CH18_Tourney_Not_Yet_Played_NOT_IN（6 行）。

- **Recipes 数据库**

(1) 显示使用了牛肉和打算的菜品。

这次使用 EXISTS 来解决这个问题，解决方案见 CH18_Recipes_Beef_And_Garlic（1 行）。

(2) 列出使用了牛肉、洋葱和胡萝卜的菜品。

这次使用 IN 来解决这个问题，但请务必小心，解决方案见 CH18_Recipes_Beef_Onion_Carrot（1 行）。

(3) 哪些菜品没有使用奶制品类食材（奶酪、黄油、牛奶）？

请使用 NOT IN 来解决这个问题，但请务必小心，确保解决方案是正确的。正确的解决方案见 CH18_Recipes_No_Dairy_RIGHT（10 行）。如果你做错了，解决方案可能类似于 CH18_Recipes_No_Dairy_WRONG（15 行）。

(4) 使用 NOT IN 解决下面两个问题：显示未在任何菜品中使用的食材；显示不包含任何菜品的菜品类型。

解决方案见 CH18_Ingredients_No_Recipe（20 行）和 CH18_Recipe_Classes_No_Recipes_NOT_IN（1 行）。

第 19 章 条件测试

“从中什么都学不到的错误才是真正的错误。”

——约翰·鲍威尔

本章涵盖如下主题：

- ❑ 条件表达式（CASE）
- ❑ 使用 CASE 解决问题
- ❑ 语句举例
- ❑ 小结
- ❑ 练习

你可能还记得，第 4 章阐述了数据和信息的差别。你将数据存储在表的行和列中，但要将数据转换为有用的信息，需要使用查询。在有些情况下，为了将数据转换为信息，需要执行复杂的计算或转换，以获得所需的结果。这种计算和转换可能很简单，如在地址标签的姓名行中加上称呼，也可能非常复杂，需要使用复杂的数学表达式。根据与最终表达式不相关的列值来计算得到所需的信息时，需要执行形如“如果……就……否则”的比较，以便编写正确的表达式。本章介绍如何解决这些类型的问题。

19.1　条件表达式（CASE）

需要检查一列的数据，以确定如何处理另一列的数据时，必须能够执行类似于下面这样的操作：如果 a 列的值为 *x*，就返回表达式 *y*，否则返回表达式 *z*。SQL 标准提供了完成这种任务的便利语法：CASE。

19.1.1　为何要使用 CASE

为了表达一组值而采用的度量衡都是人类发明的，且随时间的迁移在不断发展，因此在数据中表达值的方法有很多。距离的度量单位是英尺、英寸还是厘米？温度是摄氏温度还是华氏温度？重量的单位是磅还是千克？有些人造的体系其实很奇葩，例如，在大多数人使用的公历中，一个月可能有 28、29、30 或 31 天，而一年可能有 365 或 366 天。

你还经常会遇到这样的情形：表列中包含表示某种信息的编码值。一个极其常见的例子是使用 M 或 F 来表示性别。如果数据库中包含使用多种度量衡的数值，其中的列可能指出读数是摄氏度（C）还是华氏度（F），还可能指出距离的单位是米（M）还是英制英寸和英尺（I）。在有些情况下，单位标识包含在数据值中，例如，在货币度量衡中，负数表示“借”，而正数表示“贷”。在另外一些情况下，解码方法嵌入在数据格式中，例如，根据定义，任何月、日、年份值指的都是公历。第 5 章介绍了一种计算服务年数的简单方法，它在大多数情况下正确；我还承诺将在本章介绍一种更精确的方法。与往常一样，我说到做到！

在所有这些情况下，都必须执行某种测试，以确定该使用什么样的表达式来获得所需的信息。所有这些问题（还有其他问题）都可使用 CASE 来解决。

19.1.2 语法

我们来严肃地探索一下 CASE 表达式的语法。首先，来看看在哪些地方可以使用它。图 19-1 显示了值表达式的语法图，这是可使用 CASE 表达式的地方之一。

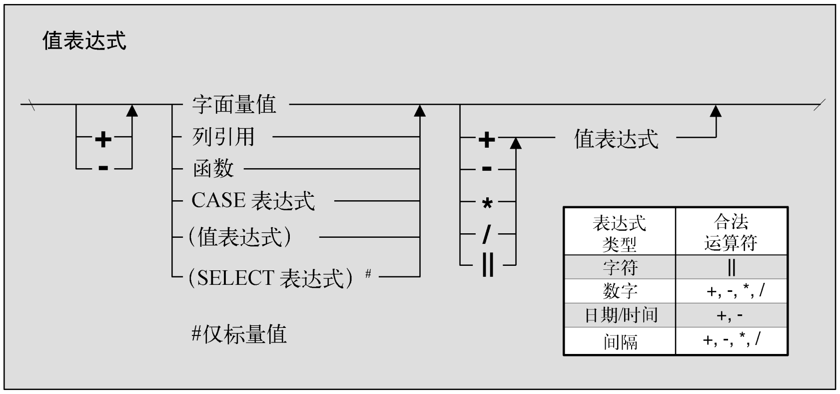


图 19-1 值表达式的语法图

乍一看，好像可使用 CASE 表达式的地方很有限，但别忘了在很多地方都可使用值表达式：可在列表中使用值表达式，还可在连接的 ON 子句、WHERE 子句或 HAVING 子句的查找条件的谓词中使用它。下面来探索 CASE 表达式的语法，图 19-2 显示了其语法图。

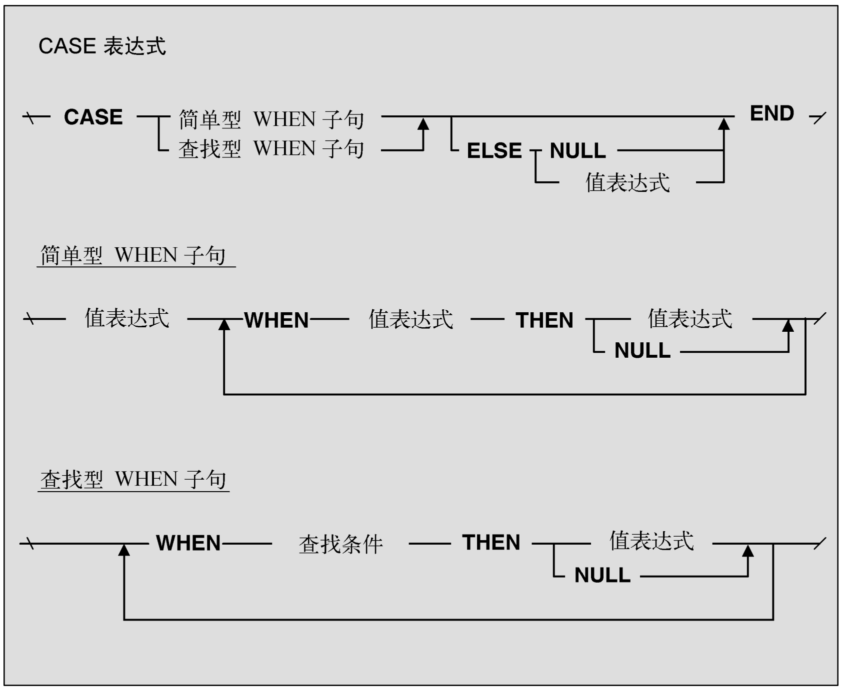


图 19-2 CASE 表达式的语法图

实际上有两种形式的 CASE 表达式。

- **简单型**：在简单型 CASE 表达式中，你使用关键字 CASE，并紧跟在这个关键字后面指定要测试的值表达式（可包含其他 CASE 表达式）。接下来，你可指定多个 WHEN/THEN 子句，将这个值表达式与多个值进行比

较，并指定在 WHEN 子句中的值表达式与关键字 CASE 后面指定的值表达式的值相同时，应返回的值表达式。请注意，这是简单的相等测试。你可在末尾使用 ELSE 子句提供一个值或 Null，这样在没有 WHEN 子句满足条件的情况下，将返回这个值。

- **查找型**：如果要执行更复杂的测试，如大于、小于、IN、BETWEEN 或 EXISTS，就必须使用查找型 CASE 表达式。在查找型 CASE 表达式中，你紧跟在关键字 CASE 后面指定一个或多个 WHEN/THEN 子句，并在关键字 WHEN 后面指定查找条件来执行复杂的比较，这包括使用子查询从另一个表中获取相关的数据。为了帮助你复习，再来看一下查找条件和谓词。请注意，无论是简单型还是查找型 CASE 表达式，数据库系统找到第一个满足条件的 WHEN 子句后，其评估过程就将结束。与简单型 CASE 表达式一样，你也可使用 ELSE 子句来指定在任何 WHEN 条件都不满足时要返回的值。图 19-3 显示了查找条件的语法图。

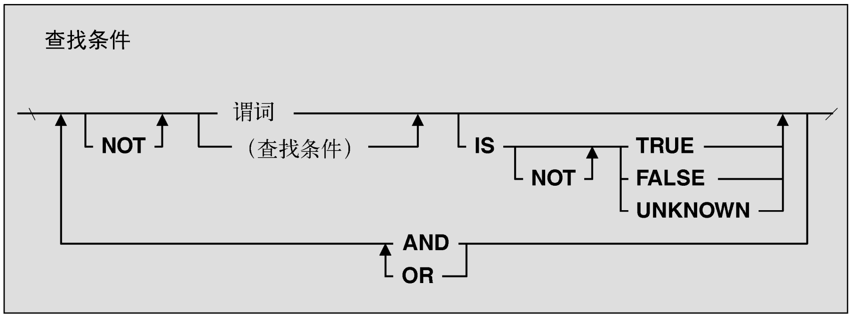


图 19-3　查找条件的语法图

请注意，你无须包含 IS [NOT] TRUE/FALSE/UNKNOWN 部分，因为数据库系统会在查找条件为真时自动计算 THEN 表达式。图 19-4 显示了谓词的语法图。

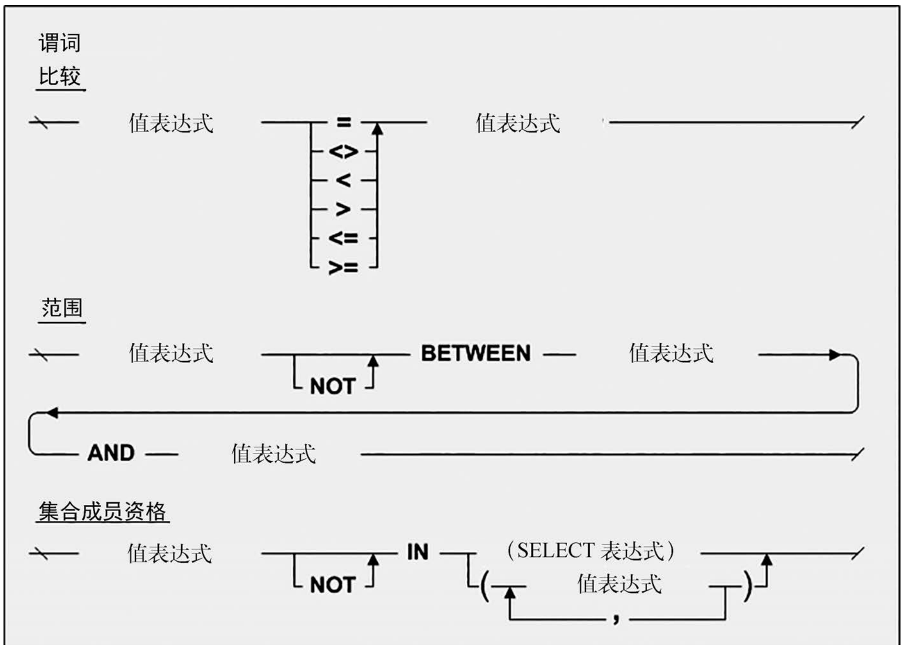


图 19-4　谓词的语法图

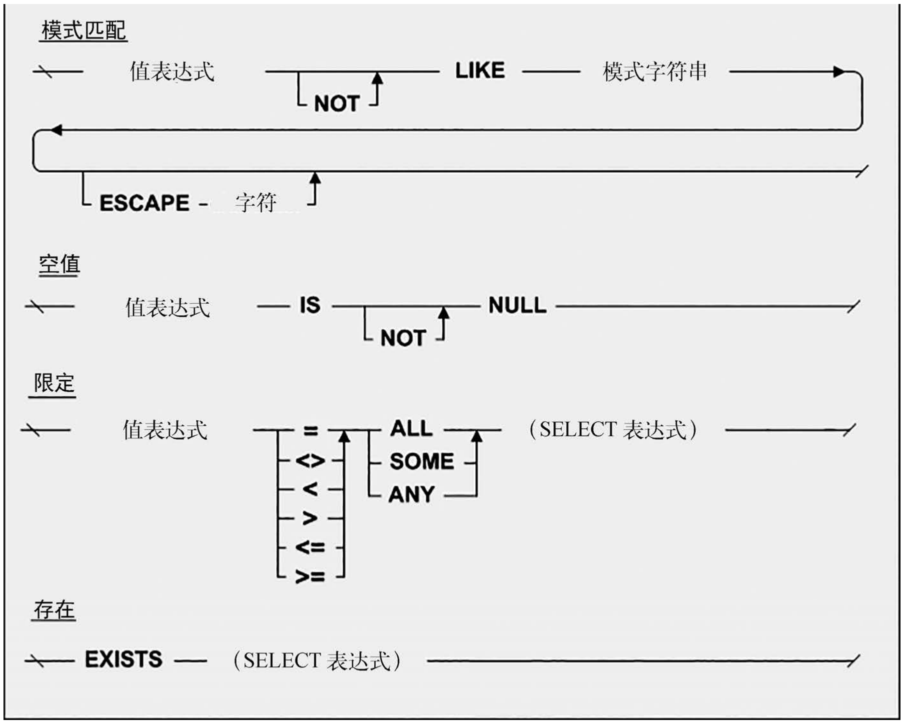


图 19-4 （续）

如你所见，在关键字 WHEN 后面指定的查找条件中，可包含的选项非常多，其中一个功能极其强大的选项是返回一个或多个值的 SELECT 表达式。

下一节将通过示例演示可以以哪些方式使用 CASE 表达式。

19.2 使用 CASE 解决问题

我们来看一个来自示例数据库 School Scheduling 的真实示例。在本节中，我将介绍如何编写简单型 CASE 表达式和查找型 CASE 表达式，还将列举一个在 WHERE 子句中使用 CASE 表达式的简单示例。

> ❖ **说明** 贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”流程。考虑到你现在对这个流程已非常熟悉，为简化流程，我在下面的示例中都将转换和整理合二为一了。

19.2.1 使用简单型 CASE 解决问题

之所以叫简单型 CASE，是因为它确实非常简单。你指定一个要测试的表达式，并在 WHEN 子句中列出要将表达式同其进行比较的值。如果它们相等，CASE 表达式将返回你在 THEN 子句中指定的表达式。你也可指定一个 ELSE 子句，这样在与任何一个值都不相等的情况下，将返回这个子句中的值表达式。

简单型 CASE 的一种常见用途是检查列中的编码值，并将其转换为更有意义的内容。假设有一个表示人的表，其中有一列指出了性别。对于这样的列，数据库设计人员可能将其指定为只能存储字母 M 或 F，以表示男性或女性。在报告中看到 M 或 F 时，大多数人明白其含义，但如果将它们转换为表示男性或女性的完整单词，不是更好吗？下面以 Students 表为例演示如何进行这种转换。

“列出学生的 ID、姓名和性别。”

显然，需要使用 CASE 和 Gender 列，并使用 WHEN 来与有效的编码进行比较，使用 THEN 来返回对应的单词。为安全起见，我们包含一个 ELSE 子句，以防遇到 Gender 列没有值的行。

转换/整理	Select student ID, stud~~ent~~ first name, stud~~ent~~ last name, ~~and~~ (CASE gender when ~~the gender code is~~ 'M'~~,~~ then ~~display~~ 'Male'~~,~~ when ~~the gender code is~~ 'F'~~,~~ then ~~display~~ 'Female'~~, and~~ else ~~display~~ 'Not Specified' END) from ~~the~~ students ~~table~~（从学生表中选择学生 ID、学生名、学生姓和学生性别，其中学生性别是这样计算得到的：检查性别列，如果其中的性别编码为 M 就显示 Male，如果性别编码为 F 就显示 Female，否则显示 Not Specified）

SQL

```sql
SELECT StudentID, StudFirstName, StudLastName,
  (CASE StudGender WHEN 'M' THEN 'Male'
   WHEN 'F' THEN 'Female'
   ELSE 'Not Specified' END) AS Gender
FROM Students
```

❖ **说明**　虽然 SQL 标准没有要求用括号将 CASE 表达式括起，但我发现，如果没有用括号将这种子句括起，Microsoft SQL Server 和 MySQL 都理解不了。添加括号可让你的意图非常清晰，且在任何情况下都没有坏处，因此我在所有的示例中都添加了括号。

另外，Microsoft Office Access 根本不支持 CASE，但提供了作用类似的内置函数（IIf）。如果你查看 Access 示例数据库中的示例，将发现我使用了 IIf 以类似的方式解决这个问题。建议你参考 Microsoft SQL Server、MySQL 和 PostgreSQL 数据库中的示例，它们遵循了 SQL 标准。

这个查询保存在了示例数据库 School Scheduling 中，名为 CH19_Student_Gender。

下面来看一个稍微复杂一点的简单型 CASE 使用示例。第 15 章演示了如何使用 Classes 和 Student Schedules 表中的数据来计算并更新每位学生的平均成绩（grade point average，GPA）。当时我指出示例查询使用了每个数据库系统特有的函数（Access 中的 NZ、SQL Server 中的 IsNull、MySQL 中的 IfNull、PostgreSQL 中的 COALESCE），这旨在避免在学生没有修完任何课程时出现除零问题。我当时指出将演示如何使用 CASE 来避免这种问题，与往常一样，我说到做到。

为了帮助你复习，这里再次列出这个问题：

“修改学生表，将平均成绩设置为学分与成绩的乘积总和再除以总学分数。”

转换/整理	Update ~~the~~ students ~~table by~~ set~~ting the~~ student GPA ~~equal to~~ = ~~the~~ (selec~~tion of the~~ sum ~~of~~ (credits ~~times~~ * grade) ~~divided by~~ / ~~the~~ sum ~~of~~ (credits) from ~~the~~ classes ~~table~~ inner join~~ed with the~~ student schedules ~~table~~ on Classes.class ID ~~in the classes table matches~~ = Student_Schedules.class ID ~~in the student schedules table~~ where ~~the~~ class status ~~is~~ = ~~complete~~ 2 and ~~the~~ student schedules ~~table~~ student ID ~~is equal to~~ = ~~the~~ students ~~table~~ student ID)［更新学生表，将学生的平均成绩设置为按如下方法计算得到的值：基于课程 ID 相同将课程表内连接到学生选课表，选择课程状态为“已结束”（2）且学生 ID 与当前学生的 ID 相同的行，计算这些行的学分与成绩的乘积总和，再除以学分总和］

SQL

```sql
UPDATE Students
SET Students.StudGPA =
    (SELECT ROUND(SUM(Classes.Credits *
        Student_Schedules.Grade) /
      SUM(Classes.Credits), 3)
     FROM Classes
     INNER JOIN Student_Schedules =
     ON Classes.ClassID
      Student_Schedules.ClassID
    WHERE (Student_Schedules.ClassStatus = 2)
     AND (Student_Schedules.StudentID =
        Students.StudentID))
```

我要做的是，在学生没有修完任何课程时，不去除以 SUM(Classes.Credits)。在这种情况下，这个总和为零，而除零通常会导致错误。由于在示例数据库中，Students 表没有 GPA 列（修改版中有，所以能够运行 UPDATE 查询），我们将这个问题作为一个简单查询来解决。图 19-5 显示了需要用到的表。

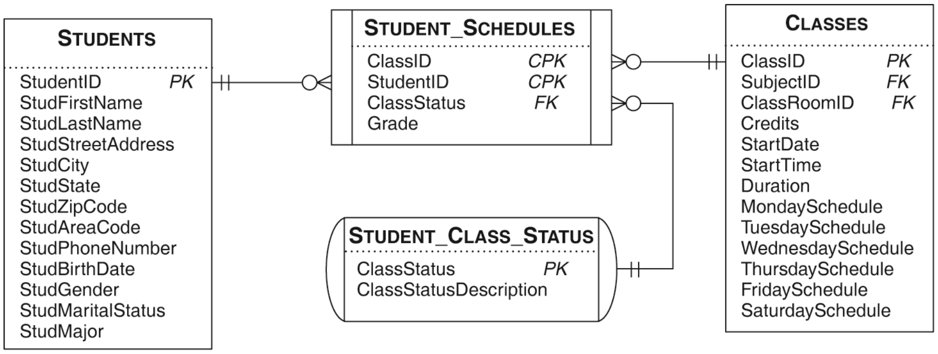


图 19-5　为计算学生平均成绩（GPA）需要用到的表

出于好玩，这次我们再严格一些，不仅要求学生修完了课程，而且要求成绩不低于 67（修完了课程但成绩为 50 时不给学分）。需要列出所有的学生，因此编写一个基于 Classes、Student_Schedules 和 Student_Class_Status 表的子查询，再将其外连接到 Students 表。

由于需要计算学分与成绩的乘积总和，再除以总学分，因此需要一个 GROUP BY 子句。为了避免除以零，需要在子查询中计算某个列包含的不为 Null 的值数（对于特定的课程，如果学生没有修完并考试通过，子查询返回的相应行的所有列都将为 Null），并使用 CASE 来检查结果是否为零。我们来编写这个查询：

“显示所有学生的 ID、名、姓及其已修完且成绩不低于 67 的课程的数量、总学分和平均成绩。”

转换/整理

Select student ID, stud~~ent~~ first name, stud~~ent~~ last name, ~~the~~ count ~~of~~ (student ID), ~~the~~ sum ~~of~~ (credits), ~~and~~ CASE ~~when the~~ count ~~of~~ (student ID) ~~is~~ WHEN 0, then ~~return~~ 0, else ~~return the~~ sum ~~of~~ (credits ~~times the~~ * grade) ~~divided by~~ / ~~the~~ sum ~~of~~ (credits) from ~~the~~ students ~~table~~ left join~~ed with the~~ (select~~ion of~~ student ID, grade, ~~and~~ credits from ~~the~~ student schedules ~~table~~ inner join~~ed with the~~ student class status ~~table~~ on Student_Schedules.class status ~~in the student schedules table matches~~ = Student_Class_Status.class status ~~in the student class status table, then~~ inner join~~ed with the~~ classes ~~table~~ on Student_Schedules.class ID ~~in the student schedules table matches~~ = Classes.class ID ~~in the classes table~~ where ~~the~~ class status description ~~equals~~ = 'Completed' and ~~the~~ grade ~~is greater than or equal to~~ >= 67) AS SClasses on Students.student ID ~~in the students table matches~~ = SClasses.student ID ~~in the selection~~ group~~ed~~ by student ID, stud~~ent~~ first name, ~~and~~ stud~~ent~~ last name（依次基于课程状态相同将学生选课表内连接到学生课程状态表、基于课程 ID 相同内连接到课程表，从课程状态描述为 Completed 且成绩不低于 67 的行中选择学生 ID、成绩和学分，生成一个结果集。将学生表左外连接到这个结果集，并按学生的 ID、名和姓分组。对于每个分组，从中选择学生的 ID、名和姓以及学生 ID 数、总学分和平均成绩。平均成绩是这样计算得到的：检查学生 ID 数，如果是零，就返回零，否则返回学分与成绩的乘积的总和再除以总学分数的结果）

SQL

```sql
SELECT Students.StudentID, Students.StudFirstName,
 Students.StudLastName,
 COUNT(SClasses.StudentID) AS NumberCompleted,
 SUM(SClasses.Credits) AS TotalCredits,
 (CASE COUNT(SClasses.StudentID)
   WHEN 0 THEN 0
   ELSE ROUND(SUM(SClasses.Credits * SClasses.Grade)
          / SUM(SClasses.Credits), 3) END) AS GPA
FROM Students LEFT OUTER JOIN
  (SELECT Student_Schedules.StudentID,
      Student_Schedules.Grade, Classes.Credits
   FROM (Student_Schedules INNER JOIN
         Student_Class_Status
         ON Student_Schedules.ClassStatus =
            Student_Class_Status.ClassStatus)
         INNER JOIN Classes
         ON Student_Schedules.ClassID =
            Classes.ClassID
   WHERE
    (Student_Class_Status.ClassStatusDescription =
          'Completed')
     AND (Student_Schedules.Grade >= 67))
       AS SClasses
ON Students.StudentID = SClasses.StudentID
GROUP BY Students.StudentID, Students.StudFirstName,
           Students.StudLastName;
```

如果学生没有修完并通过一门课程的考试，则其课程数、学分和平均成绩都为零。为了避免除零问题，我采取了符合 SQL 标准的做法，不需要使用数据库系统中的非标准函数。这个查询保存到了示例数据库 School Scheduling 中，名为 CH19_Student_GPA_Avoid_0_Passed。

19.2.2　使用查找型 CASE 解决问题

你可能认为简单型 CASE 使用起来有点别扭，但查找型 CASE 使用起来肯定很有趣。使用简单型 CASE 时，只能执行简单的相等比较（这可能就是它被称为简单型的原因），但使用查找型 CASE 时，可根据多个列乃至子查询执行多种复杂的比较。大致而言，在连接的 ON 子句、WHERE 子句或 HAVING 子句中可使用的任何查找条件，都可在查找型 CASE 中使用。下面来解决两个要求使用查找型 CASE 的问题。

第 5 章演示了如何计算员工到特定日期服务了多少个整年，方法是计算聘用日期与目标日期相隔多少天，再除以每年的天数 365。我当时提醒过，这种计算方法不精确，因为他没有考虑这两个日期之间可能有闰年。事实上，每有一个闰年，就有 1 天的误差，但仅当聘用日期和目标日期的月和日很接近时才会影响结果，因此在大多数情况下，除以 365 得到的答案是正确的。

我当时还承诺过，将演示如何使用 CASE 精确地计算答案。大致而言，需要将目标年份减去聘用年份，并检查聘用月和日是否落在目标月和日的后面，如果是这样的，就将结果减 1。换而言之，如果在目标年份，员工的入职周年日还没到，他就还没有又服务了一年（你要计算的是服务了多少个整年）。

因此，可先将两个年份相减，再检查聘用月是否大于目标月（如果这两个月份相同，再检查聘用日是否大于目标日），如果答案是肯定的，就将前面计算得到的差减 1。计算公式类似于下面这样：

```
Years of service = ((year of target) - (year of date hired)) -
           (If month of date hired < month of target then 0
           If month of date hired > month of target then 1
           If month of date hired = month of target and
              day of date hired > target then 1
           Else 0)
```

服务年数 = [(目标年份) – (聘用年份)] – (如果聘用月份小于目标月份，则为 0；如果聘用月份大于目标月份，则为 1；如果聘用月份等于目标月份且聘用日大于目标日，则为 1；否则为 0)

我需要做的是，将上述计算公式中的“如果……则……否则”部分转换为一个查找型 CASE 表达式，如下所示：

“列出所有教员的 ID、名、姓、聘用日期以及到 2017 年 10 月 1 日服务了多少个整年，并按姓和名排序。”

转换/整理	Select staff ID, staff first name, staff last name, date hired, ~~and calculate the~~ year ~~of the date~~ (2017-10-1) ~~minus the~~ year ~~of the~~ (date hired) ~~minus~~ – (CASE when ~~the~~ month ~~of the~~ (date hired) ~~is less than~~ < 10 then 0, when ~~the~~ month ~~of the~~ (date hired) ~~is greater than~~ > 10 then 1, ~~and~~ when ~~the~~ day ~~of the~~ (date hired) ~~is greater than~~ > 1 then 1 else 0 END) as ~~the~~ length of service from ~~the~~ staff ~~table~~ order~~ed~~ by staff last name, ~~and~~ staff first name（从教员表中选择教员的 ID、名、姓、聘用日期以及服务了多少个整年，并按姓和名排序。其中服务了多少个整年是这样计算得到的：将 2017-10-1 的年份部分减去聘用年份，再根据情况减去一个值，即如果聘用月份小于 10 就减去 0，如果聘用月份大于 10 就减去 1，如果聘用日大于 1 就减去 1，否则减去 0）

SQL

```
SELECT StaffID, StfFirstName, StfLastname,
   YEAR(CAST('2017-10-01' As Date))
   - YEAR(DateHired) -
   - CASE WHEN Month(DateHired) < 10
          THEN 0
          WHEN Month(DateHired) > 10
          THEN 1
          WHEN Day(DateHired) > 1
          THEN 1
          ELSE 0 END) AS LengthOfService
FROM Staff
ORDER BY StfLastName, StfFirstName;
```

> ❖ **说明**　几乎所有数据库系统都内置了函数 YEAR、MONTH 和 DAY，可用于获取日期的相应部分（在 Oracle 中，需要使用函数 EXTRACT），因此我决定在这个示例中使用这些函数，虽然 SQL 标准没有专门定义它们。另外，由于这里的目标日期是已知的，因此我在 CASE 语句中直接使用了 10 和 1。如果你使用的是从数据库系统获取的诸如当前日期等日期，就必须在比较前提取其中的日和月份部分。

我将这个查询保存到了示例数据库 School Scheduling 中，并将其命名为 CH19_Length_Of_Service。如果你执行这个查询和 CH05_Length_Of_Service，将发现在这个使用 CASE 的精确查询的结果中，Jeffrey Smith 的服务年数少了 1 年。Jeffrey Smith 的聘用日期为 1991 年 10 月 6 日，其中的日子部分比目标日晚了 5 天。出现这种误差的原因是，在 1991 年到 2017 年之间，闰年数超过了 6 个。

你可能会问：为何不显式地测试月份为 10？别忘了，数据库系统依次评估不同的 WHEN 子句，直到找到一个为真的。如果月份不小于 10 且不大于 10，那么它必然等于 10，因此没必要显式测试这一点。

再来看一个使用查找型 CASE 的示例。假设你要生成一个邮寄清单，而表中有性别和婚姻状况信息，但没有称呼（Mr.、Mrs.等）。我们通过使用 CASE 检查既有列来生成称呼。由于需要检查多个列，因此必须使用查找型 CASE。

“生成一个学生邮寄清单，其中包含生成的称呼、名、姓、街道地址、城市、州和邮政编码。”

转换/整理	Select (CASE when ~~the~~ gender ~~is male~~ = ‘M’ then ~~return~~ ‘Mr.’, when ~~the~~ marital status ~~is single~~ = ‘S’ then ~~return~~ ‘Ms.’ else ~~return~~ ‘Mrs.’ END) ~~concatenated with~~ || stud~~ent~~ first name ~~concatenated with a space~~ || ‘ ’ ~~and~~ stud~~ent~~ last name as the name line, stud~~ent~~ street address as ~~the~~ street line, student city ~~concatenated with a comma and a space~~ || ‘, ’~~then concatenated with~~ || stud~~ent~~ state ~~and two spaces~~ || ‘ ’, ~~then concatenated with~~ || stud~~ent~~ ZIP code as ~~the~~ city line from ~~the~~ students ~~table~~（从学生表中选择姓名行、街道地址、城市行。姓名行是将称呼、名、一个空格和姓拼接而成的，而其中的称呼是这样生成的：如果性别为 M，就返回“Mr.”；如果婚姻状况为 S，就返回“Ms. ”；否则返回“Mrs.”。城市行是将城市、一个逗号和一个空格、州、两个空格以及邮政编码拼接而成的）

SQL

```
SELECT
    (CASE WHEN StudGender = 'M' THEN 'Mr. '
          WHEN StudMaritalStatus = 'S' THEN 'Ms. '
          ELSE 'Mrs. ' END)
   || StudFirstName || ' ' || StudLastName
      AS NameLine,
   StudStreetAddress AS StreetLine,
   StudCity || ', ' || StudState || '  ' ||
   StudZipCode AS CityLine
FROM Students
```

所有男学生的称呼都为 Mr.，因此没有必要检查男学生的婚姻状况。如果学生不是男的，就没有必要再检查性别，因为只能是女的（F）。最后，如果女学生是单身，称呼为 Ms.，否则为 Mrs.［这包括已婚（M）、离异（D）和丧偶（W）］。只要巧妙地编写 WHEN/THEN 对，就无须检查每种可能的组合。我将这个查询保存到了示例数据库 School Scheduling 中，并将其命名为 CH19_Student_Mailing_List。

19.2.3　在 WHERE 子句中使用 CASE

为确保全面，我们来看看如何在 WHERE（或 HAVING）子句中使用 CASE。坦率地说，我想不出使用其他方式编写不出更清晰的谓词的例子。我说过很多次，可以做并不意味着该做！话虽如此，我们还是来试试这样做，看看结果是什么样的。

“列出所有的男学生。”

转换/整理	Select student ID, stud~~ent~~ first name, stud~~ent~~ last name, ~~and~~ ‘Male’ as gender from ~~the~~ students ~~table~~ where ‘Male’ ~~equals~~ = (CASE when ~~the~~ stud~~ent~~ gender ~~is~~ ‘M’ then ~~return~~ ‘Male’ else ~~return~~ ‘Nomatch’ END)［对于学生表中的每行，检查其性别列并返回相应的值（如果为 M 就返回 Male，否则返回 Nomatch）再检查 Male 是否与返回的值相等，如果相等，就从该行选择学生的 ID、名、姓和 Male］

SQL	`SELECT StudentID, StudFirstName, StudLastName, 'Male' AS Gender FROM Students WHERE ('Male' = (CASE StudGender WHEN 'M' THEN 'Male' ELSE 'Nomatch' END));`

```
SQL          SELECT StudentID, StudFirstName, StudLastName,
                    'Male' AS Gender
             FROM Students
             WHERE ('Male' = (CASE StudGender
                              WHEN 'M' THEN 'Male'
                              ELSE 'Nomatch' END));
```

这里的诀窍是，由于这个请求说要获取男学生，因此我完全按字面意思解释。为了生成值“Male”，我使用 CASE 在性别列的值为“M”时返回这个单词。坦率地说，像下面这样指定查找条件要容易得多：

```
WHERE Gender = 'M'
```

这种用法不是很有用，但我还是将这个查询保存到了示例数据库 School Scheduling 中，并将其命名为 CH19_Male_Students。

19.3　语句举例

至此，你知道了如何使用 CASE 编写查询，还知道了使用 CASE 可解决的一些问题类型，下面来看一些示例，它们都使用了简单型 CASE 或查找型 CASE。这些示例都来自示例数据库，演示了如何在值表达式中使用 CASE 来执行条件测试。

❖ **说明**　本书前言提醒过，在你使用的数据库系统中，行的排列顺序不一定与本书示例中显示的相同——除非使用 ORDER BY 子句。即便指定了排序方式，在数据库系统返回的结果中，对于 ORDER BY 子句中未包含的列，根据它们的排列顺序也可能不同，这是因为优化方法因数据库系统而异。

如果你在 Microsoft SQL Server 中运行示例，则从视图中选择行时，将不会遵循其中的 ORDER BY 子句指定的排列顺序。要遵循 ORDER BY 子句指定的排列顺序，必须打开视图的设计，并在显示界面中执行它。

另外，当你使用 GROUP BY 子句时，数据库系统返回的结果集看起来像是根据该子句中指定的列进行了排序。这是因为有些优化器首先会在内部对数据排序，以提高 GROUP BY 的处理速度。别忘了，如果要按特定的顺序排列，必须使用 ORDER BY 子句。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库中，名称以 CH19 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

❖ **说明**　别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录 B。这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。别忘了，对于包含 ORDER BY 子句的 SQL Server 视图，必须先在“设计”模式下打开再执行它，结果才会按指定顺序排列；如果你使用 SELECT *获取视图内容，SQL Server 将不会按 ORDER BY 子句指定的顺序排列结果集。

- **Sales Orders 数据库**

“列出所有商品并指出各种商品在 2071 年 12 月是否有顾客购买过。”

转换/整理	Select product number, product name, ~~and~~ (CASE when product number ~~is~~ in ~~the~~ (select~~ion of~~ product number from ~~the~~ order details ~~table inner~~ join~~ed with the~~ orders ~~table~~ on Orders.order number ~~in the orders table matches~~ = Order_Details.order number ~~in the order details table~~ where ~~the~~ order date ~~is~~ between ‘2017-12-01 and ‘2017-12-31’) then ~~return~~ ‘Ordered’ else ~~return~~ ‘Not Ordered’ END) as product ordered from ~~the~~ products ~~table~~（从商品表中选择商品编号、商品名和“是否有顾客购买”，其中是否“是否有顾客购买”是这样确定的：基于订单号相同将订单详情表内连接到订单表，从下单日期在 2017-12-01 和 2017-12-31 之间的行中选择订单号，生成一个列表；如果当前行的订单号在这个列表中，就返回 Ordered，否则返回 Not Ordered）

SQL

```
SELECT ProductNumber, ProductName,
  (CASE WHEN Products.ProductNumber IN
     (SELECT Order_Details.ProductNumber
      FROM Order_Details INNER JOIN Orders
      ON Orders.OrderNumber =
         Order_Details.OrderNumber
      WHERE (Orders.OrderDate BETWEEN
                CAST('2017-12-01' AS Date) AND
                CAST('2017-12-31' AS Date)))
        THEN 'Ordered'
        ELSE 'Not Ordered' END) AS
        ProductOrdered
FROM Products;
```

CH19_Products_Ordered_Dec_2017（40 行）

ProductNumber	ProductName	ProductOrdered
1	Trek 9000 Mountain Bike	Ordered
2	Eagle FS-3 Mountain Bike	Ordered
3	Dog Ear Cyclecomputer	Ordered
4	Victoria Pro All Weather Tires	Not Ordered
5	Dog Ear Helmet Mount Mirrors	Ordered
6	Viscount Mountain Bike	Ordered
7	Viscount C-500 Wireless Bike Computer	Ordered
8	Kryptonite Advanced 2000 U-Lock	Ordered
	<< 其他行 >>	

“显示所有商品及其销量等级。销量等级这样确定：销量不超过 200 为 Poor（差）；超过 200 但不超过 500 为 Average（一般）；超过 500 但不超过 1000 为 Good（良好）；超过 1000 为 Excellent（优秀）。”

转换/整理

Select product number, product name, ~~and~~ (CASE when ~~the~~ (selec~~tion of the~~ sum ~~of~~ (quantity ordered) from ~~the~~ order details ~~table~~ where ~~the~~ Order_Details.product number ~~in the order details table equals~~ = the Products.product number ~~in the products table) is less than or equal to~~ <= 200 then ~~return~~ 'Poor' when ~~the~~ (selec~~tion of the~~ sum ~~of~~ (quantity ordered) from ~~the~~ order details ~~table~~ where ~~the~~ Order_Details.product number ~~in the order details table equals~~ = ~~the~~ Products.product number ~~in the products table~~) ~~is less than or equal to~~ <= 500 then ~~return~~ 'Average' when ~~the~~ (selec~~tion of the~~ sum ~~of~~ (quantity ordered) from ~~the~~ order details ~~table~~ where ~~the~~ Order_Details.product number ~~in the order details table equals~~ = ~~the~~ Products. product number ~~in the products table~~) ~~is less than or equal to~~ <= 1000 then ~~return~~ 'Good' else ~~return~~ 'Excellent' END) as sales quality from ~~the~~ products ~~table~~（从商品表中选择商品编号、商品名和销量等级，其中销量等级是这样确定的：在订单详情表中，选择商品编号与当前商品的编号相同的行，计算这些行的订购数量总和，如果不超过 200 就返回 Poor，如果不超过 500 就返回 Average，如果不超过 1000 就返回 Good，否则返回 Excellent）

SQL

```
SELECT ProductNumber, ProductName,
  (CASE WHEN
          (SELECT SUM(QuantityOrdered)
           FROM Order_Details
           WHERE (Order_Details.ProductNumber =
               Products.ProductNumber)) <= 200
         THEN 'Poor'
         WHEN
          (SELECT SUM(QuantityOrdered)
           FROM Order_Details
           WHERE (Order_Details.ProductNumber =
               Products.ProductNumber)) <= 500
        THEN 'Average'
        WHEN
          (SELECT SUM(QuantityOrdered)
           FROM Order_Details
           WHERE (Order_Details.ProductNumber =
              Products.ProductNumber)) <= 1000
        THEN 'Good'
        ELSE 'Excellent' END) AS SalesQuality
FROM Products
```

CH19_Products_And_SalesQuality（40 行）

ProductNumber	ProductName	SalesQuality
1	Trek 9000 Mountain Bike	Excellent
2	Eagle FS-3 Mountain Bike	Poor
3	Dog Ear Cyclecomputer	Poor
4	Victoria Pro All Weather Tires	Excellent
5	Dog Ear Helmet Mount Mirrors	Poor
6	Viscount Mountain Bike	Good
7	Viscount C-500 Wireless Bike Computer	Average
8	Kryptonite Advanced 2000 U-Lock	Poor
	<< 其他行 >>	

❖ **说明**　虽然问题中说的是诸如“超过 200 但不超过 500”等条件，但在每个 WHEN 子句中无须指定超过部分，因为前一个 WHEN 子句的不超过条件已经不满足（如果销量不是<= 200，那么肯定是超过 200）。如果能够使用简单型 CASE 就好了，这样只需运行子查询一次，但没法这样做，因为这里的测试不是相等比较。如果数据库系统包含智能优化器，将发现全部三个子查询都相同，进而对于每一行只执行子查询一次。

- **Entertainment Agency 数据库**

“列出每个演唱组合并指出它在 2017 年圣诞节（12 月 25 日）是否有演出合约。”

转换/整理　Select entertainer ID, ent~~ertainer~~ stage name ~~and~~ (CASE when entertainer ID ~~is~~ in ~~the~~ (selecti~~on of the~~ entertainer ID from ~~the~~ engagements ~~table~~ where ‘2017-12-25’ ~~is~~ between start date and end date) then ~~return~~ ‘Booked’ else ~~return~~ ‘Not Booked’ END) as booked Xmas 2017 from ~~the~~ entertainers ~~table~~（从演唱组合表中选择演唱组合的 ID 和艺名以及 2017 年圣诞节是否有演出合约。2017 年圣诞节是否有演出合约是这样确定的：在演出合约表中，从 2017-12-25 在其开始日期和结束日期之间的行中选择演唱组合 ID，生成一个列表；如果当前演唱组合的 ID 在这个列表中，就返回 Booked，否则返回 Not Booked）

SQL
```
SELECT EntertainerID, EntStageName,
    (CASE WHEN EntertainerID IN
             (SELECT EntertainerID
              FROM Engagements
              WHERE CAST('2017-12-25' AS Date)
                    BETWEEN StartDate AND
                    EndDate)
          THEN 'Booked'
          ELSE 'Not Booked' END) AS BookedXmas2017
FROM Entertainers;
```

CH19_Entertainers_Booked_Xmas_2017（13 行）

EntertainerID	EntStageName	BookedXmas2017
1001	Carol Peacock Trio	Booked
1002	Topazz	Not Booked
1003	JV & the Deep Six	Booked
1004	Jim Glynn	Not Booked
1005	Jazz Persuasion	Booked
1006	Modern Dance	Booked
1007	Coldwater Cattle Company	Not Booked
1008	Country Feeling	Not Booked
	<< 其他行 >>	

❖ **说明**　别忘了，演出合约表包含开始日期和结束日期，因此你需要使用 BETWEEN 找出 2017 年 12 月 25 日在其开始日期和结束日期之间的演出合约。

“在 WHERE 子句中使用查找型 CASE 找出喜欢爵士乐（Jazz）但不喜欢标准歌曲（Standards）的顾客。”

转换/整理

Select customer ID, cust~~omer~~ first name ~~and~~ cust~~omer~~ last name from ~~the~~ customers ~~table~~ where ~~true~~ 1 ~~equals~~ = (CASE when customer ~~is~~ not in ~~the~~ (selec~~tion of the~~ customer ID from ~~the~~ musical preferences ~~table~~ inner join~~ed with the~~ musical styles ~~table~~ on Musical_Preferences.style ID ~~in the musical preferences table equals~~ = Musical_Styles.style ID ~~in the musical styles table~~ where style name ~~equals~~ = ‘Jazz’) then ~~return~~ 0 when customer ~~is~~ in ~~the~~ (selec~~tion of the~~ customer ID from ~~the~~ musical preferences ~~table~~ inner join~~ed with the~~ musical styles ~~table~~ on Musical_Preferences.style ID ~~in the musical preferences table equals~~ = Musical_Styles.style ID ~~in the musical styles table~~ where style name ~~equals~~ = ‘Standards’) then ~~return~~ 0 else ~~return~~ 1 END)（在顾客表中，从满足条件“1＝返回值”的行中选择顾客的 ID、名和姓。“返回值”是这样确定的：基于风格 ID 相同将音乐喜好表内连接到音乐风格表，生成一个结果集。在上述结果集中，从风格名为 Jazz 的行中选择顾客 ID，生成一个列表；如果当前顾客的 ID 不在这个列表中，就返回 0。在上述结果集中，从风格名为 Standards 的行中选择顾客 ID，生成一个列表，如果当前顾客的 ID 在这个列表中就返回 0，否则返回 1）

SQL

```
SELECT CustomerID, CustFirstName, CustLastName
FROM Customers
WHERE (1 =
       (CASE WHEN CustomerID NOT IN
              (SELECT CustomerID
               FROM Musical_Preferences
               INNER JOIN Musical_Styles
               ON Musical_Preferences.StyleID =
                  Musical_Styles.StyleID
               WHERE Musical_Styles.StyleName =
                       'Jazz')
                       THEN 0
                       WHEN CustomerID IN
                        (SELECT CustomerID
                         FROM Musical_Preferences
                         INNER JOIN Musical_Styles
                        ON Musical_Preferences.StyleID =
                           Musical_Styles.StyleID
                        WHERE Musical_Styles.StyleName =
                              'Standards')
                       THEN 0 ELSE 1 END));
```

❖ **说明** 虽然要求的是找出喜欢爵士乐但不喜欢标准歌曲的顾客，但别忘了，找到第一个结果为真的 WHEN/THEN 子句后，对 CASE 表达式的评估将就此结束。有鉴于此，我按相反的逻辑编写测试，先排除不喜欢爵士乐的顾客，再排除喜欢标准歌曲的顾客。如果我先指定找出喜欢爵士乐的顾客的测试，将选择所有喜欢爵士乐的顾客，而不再执行后续检查，找出其中不喜欢标准歌曲的顾客。

CH19_Customers_Jazz_Not_Standards（2 行）

CustomerID	CustFirstName	CustLastName
10010	Zachary	Ehrlich
10013	Estella	Pundt

- **School Scheduling 数据库**

“显示所有全职教员加薪后的薪水。加薪规则如下：给讲师加薪 5%；给副教授加薪 4%；给教授加薪 3.5%。”

转换/整理

Select staff ID, st~~aff~~ first name, st~~aff~~ last name, title, status, salary, ~~and~~ (CASE ~~when~~ title ~~is~~ WHEN ‘Instructor’ then ~~return~~ salary ~~times~~ * 1.05 when ‘Associate Professor’ then ~~return~~ salary ~~times~~ * 1.04 when ‘Professor’ then ~~return~~ salary ~~times~~ * 1.035 else ~~return~~ salary END) as new salary from ~~the~~ staff ~~table~~ inner join~~ed with the~~ faculty ~~table~~ on Staff.staff ID ~~in the staff table equals~~ = Faculty. staff ID ~~in the faculty table~~ where status ~~equals~~ = ‘Full Time’（基于教员 ID 相同将教员表内连接到全体教员表，从状态为 Full Time 的行中选择教员的 ID、名、姓、职称、状态、薪水和加薪后的薪水。加薪后的薪水是这样计算得到的：如果教员的职称为 Instructor，就返回当前薪水与 1.05 的乘积；如果教员的职称为 Associate Professor，就返回当前薪水与 1.04 的乘积；如果教员的职称为 Professor，就返回当前薪水与 1.035 的乘积）

SQL

```
SELECT StaffID, StfFirstName, StfLastname, Title,
   Status, Salary,
   (CASE Title
      WHEN 'Instructor'
      THEN ROUND(Salary * 1.05, 0)
```

```
        WHEN 'Associate Professor'
        THEN ROUND(Salary * 1.04, 0)
        WHEN 'Professor'
        THEN ROUND(Salary * 1.035, 0)
        ELSE Salary END) AS NewSalary
FROM Staff INNER JOIN Faculty
 ON Staff.StaffID = Faculty.StaffID
WHERE Faculty.Status = 'Full Time';
```

CH19_FullTime_Instructor_Raises（22 行）

StaffID	StfFirstName	StfLastName	Title	Status	Salary	NewSalary
98005	Suzanne	Viescas	Instructor	Full	$44,000.00	$46,200.00
98007	Gary	Hallmark	Associate Professor	Full	$53,000.00	$55,120.00
98011	Ann	Patterson	Instructor	Full	$45,000.00	$47,250.00
98012	Robert	Brown	Instructor	Full	$49,000.00	$51,450.00
98013	Deb	Waldal	Instructor	Full	$44,000.00	$46,200.00
98014	Peter	Brehm	Professor	Full	$60,000.00	$62,100.00
98019	Mariya	Sergienko	Instructor	Full	$45,000.00	$47,250.00
98020	Jim	Glynn	Instructor	Full	$45,000.00	$47,250.00
			<< 其他行 >>			

“列出所有学生及其注册的课程、各门课程的成绩及其字母表示。”

❖ **说明**　我将使用美国通用的转换方案：97 到 100 为 A+，93 到 96.99 为 A，90 到 92.99 为 A–，以此类推（每个字母分 3 个等级，对应于 10 分的区间），最后 60 到 62.99 为 D–，其他为不及格（F）。

转换/整理

Select student ID stud~~ent~~ first name, stud~~ent~~ last name, class ID start date, subject code, subject name, grade, ~~and~~ (CASE when ~~the~~ grade ~~is~~ between 97 and 100 then ~~return~~ ‘A+’, when ~~the~~ grade ~~is~~ between 93 and 96.99 then ~~return~~ ‘A’, when ~~the~~ grade ~~is~~ between 90 and 92.99 then ~~return~~ ‘A-’, when ~~the~~ grade ~~is~~ between 87 and 89.99 then ~~return~~ ‘B+’, when ~~the~~ grade ~~is~~ between 83 and 86.99 then ~~return~~ ‘B’, when ~~the~~ grade ~~is~~ between 80 and 82.99 then ~~return~~ ‘B-’, when ~~the~~ grade ~~is~~ between 77 and 79.99 then ~~return~~ ‘C+’, when ~~the~~ grade ~~is~~ between 73 and 76.99 then ~~return~~ ‘C’, when ~~the~~ grade ~~is~~ between 70 and 72.99 then ~~return~~ ‘C-’, when ~~the~~ grade ~~is~~ between 67 and 69.99 then ~~return~~ ‘D+’, when ~~the~~ grade ~~is~~ between 63 and 66.99 then ~~return~~ ‘D’, when ~~the~~ grade ~~is~~ between 60 and 62.99 then ~~return~~ ‘D-’, else ~~return~~ ‘F’ END) as letter grade from ~~the~~ students ~~table~~ inner join~~ed with the~~ student schedules ~~table~~ on Students.student ID ~~in the students table equals~~ = Student_Schedules.student ID ~~in the student schedules table, then~~ inner join~~ed with the~~ classes ~~table~~ on Student_Schedules.class ID ~~in the student schedules table equals~~ = Classes.class ID ~~in the classes table, then~~ inner join~~ed with the~~ subjects ~~table~~ on Classes.subject ID ~~in the classes table equals~~ = Subjects.subject ID ~~in the subjects table, and then finally~~ inner join~~ed with the~~ student class status ~~table~~ on Student_Schedules.class status ~~in the student schedules table equals~~ = Student_Class_Status.class status ~~in the student class status table~~ where class status description ~~equals~~ = ‘Completed’（依次基于学生 ID 相同将学生表内连接到学生选课表、基于课程 ID 相同内连接到课程表、基于科目 ID 相同内连接到科目表、基于课程状态相同内连接到学生课程状态表，从课程状态描述为 Completed 的行中选择学生 ID、学生名、学生姓、开始日期、科目编码、科目名、成绩和成绩的字母表示。成绩的字母表示是这样确定的：如果成绩为 97~100，就返回 A+；如果为 93~96.99，就返回 A；如果为 90~92.99，就返回 A–；如果为 87~89.99，就返回 B+；如果为 83~86.99，就返回 B；如果为 80~82.99，就返回 B–；如果为 77~79.99，就返回 C+；如果为 73~76.99，就返回 C；如果为 70~72.99，就返回 C–；如果为 67~69.99，就返回 D+；如果为 63~66.99，就返回 D；如果为 60~62.99，就返回 D–；否则返回 F）

SQL

```
SELECT Students.StudentID, Students.StudFirstName,
Students.StudLastName, Classes.ClassID,
Classes.StartDate, Subjects.SubjectCode,
Subjects.SubjectName, Student_Schedules.Grade,
```

```
     (CASE WHEN Grade BETWEEN 97 AND 100 THEN 'A+'
           WHEN Grade BETWEEN 93 AND 96.99 THEN 'A'
           WHEN Grade BETWEEN 90 AND 92.99 THEN 'A-'
           WHEN Grade BETWEEN 87 AND 89.99 THEN 'B+'
           WHEN Grade BETWEEN 83 AND 86.99 THEN 'B'
           WHEN Grade BETWEEN 80 AND 82.99 THEN 'B-'
           WHEN Grade BETWEEN 77 AND 79.99 THEN 'C+'
```

```
                WHEN Grade BETWEEN 73 AND 76.99 THEN 'C'
                WHEN Grade BETWEEN 70 AND 72.99 THEN 'C-'
                WHEN Grade BETWEEN 67 AND 69.99 THEN 'D+'
                WHEN Grade BETWEEN 63 AND 66.99 THEN 'D'
                WHEN Grade BETWEEN 60 AND 62.99 THEN 'D-'
                ELSE 'F' END) AS LetterGrade
FROM (((Students INNER JOIN Student_Schedules
ON Students.StudentID = Student_Schedules.StudentID)
INNER JOIN Classes
ON Student_Schedules.ClassID = Classes.ClassID)
INNER JOIN Subjects
ON Classes.SubjectID = Subjects.SubjectID)
INNER JOIN Student_Class_Status
ON Student_Schedules.ClassStatus =
   Student_Class_Status.ClassStatus
WHERE Student_Class_Status.ClassStatusDescription =
       'Completed';
```

CH19_Students_Classes_Letter_Grades（68 行）

Student ID	StudFirst Name	StudLast Name	ClassID	StartDate	Subject Code	Subject Name	Grade	Letter Grade
1001	Kerry	Patterson	2907	2017-09-11	MAT 097	Elementary Algebra	67.33	D+
1001	Kerry	Patterson	3085	2017-09-11	HIS 111	U.S. History to 1877	87.14	B+
1002	David	Hamilton	1156	2017-09-11	ENG 101	Composition - Fundamentals	86.33	B
1002	David	Hamilton	1500	2017-09-11	MUS 100	Music in the Western World	85.72	B
1002	David	Hamilton	2889	2017-09-11	MAT 080	Preparatory Mathematics	68.22	D+
1003	Betsy	Stadick	1156	2017-09-11	ENG 101	Composition Fundamentals	71.09	C–

<< 其他行 >>

- **Bowling League 数据库**

“列出所有的投球手及其等级。等级是这样确定的：平均原始得分低于 140 为差（Fair）；不低于 140 但低于 160 为一般（Average）；不低 160 但低于 185 为良（Good）；不低于 185 为优（Excellent）。”

转换/整理

Select bowler ID, bowler last name, bowler first name, ~~the~~ av~~erage of~~ (raw score), ~~and~~ (CASE when ~~the~~ av~~erage of~~ (raw score) ~~is less than~~ < 140, then ~~return~~ ‘Fair’, when the av~~erage of~~ (raw score) ~~is less than~~ < 160, then ~~return~~ ‘Average’, when ~~the~~ av~~erage of~~ (raw score) ~~is less than~~ < 185, then ~~return~~ ‘Good’, else ~~return~~ ‘Excellent’ END) as bowler rating from ~~the~~ bowlers ~~table~~ inner join~~ed with the~~ bowler scores ~~table~~ on Bowlers.bowler ID ~~in the bowlers table equals~~ = Bowler_Scores.bowler ID ~~in the bowler scores table~~ grouped ~~by~~ bowler ID, bowler last name, ~~and~~ bowler first name（基于投球手 ID 相同将投球手表内连接到投球手得分表，按投球手 ID、投球手姓、投球手名分组；从每个分组中选择投球手 ID、投球手姓、投球手名、投球手平均原始得分和投球手等级。投球手等级是这样确定的：如果平均原始得分低于 140，就返回 Fair；如果不低于 140 但低于 160，就返回 Average；如果不低于 160 但低于 185，就返回 Good；否则返回 Excellent）

SQL

```
SELECT Bowlers.BowlerID, Bowlers.BowlerLastName,
       Bowlers.BowlerFirstName,
 CAST(AVG(RawScore) AS Int) AS BowlerAverage,
 (CASE
  WHEN CAST(AVG(Bowler_Scores.RawScore) AS Int)
     < 140
  THEN 'Fair'
  WHEN CAST(AVG(Bowler_Scores.RawScore) AS Int)
     < 160
  THEN 'Average'
  WHEN CAST(AVG(Bowler_Scores.RawScore) AS Int)
     < 185
  THEN 'Good'
```

```
  ELSE 'Excellent' END) AS BowlerRating
FROM Bowlers INNER JOIN Bowler_Scores
ON Bowlers.BowlerID = Bowler_Scores.BowlerID
GROUP BY Bowlers.BowlerID, Bowlers.BowlerLastName,
           Bowlers.BowlerFirstName;
```

CH19_Bowler_Ratings（32 行）

BowlerID	BowlerLastName	BowlerFirstName	BowlerAverage	BowlerRating
1	Fournier	Barbara	148	Average
2	Fournier	David	156	Average
3	Kennedy	John	165	Good
4	Sheskey	Sara	141	Average
5	Patterson	Ann	149	Average
6	Patterson	Neil	158	Average
7	Viescas	David	167	Good
8	Viescas	Stephanie	142	Average
		<< 其他行 >>		

❖ **说明**　别忘了，数据库系统在发现第一个条件为真的 WHEN/THEN 子句后停止评估，因此这里不需要在检查<的同时检查>=。

“显示所有的联赛及其场次详情（对于还没有举行的比赛场次，显示 Not Played Yet）。”

转换/整理

Select tourney ID, tourney date, tourney location, ~~and~~ (CASE when the match ID is ~~empty~~ NULL then ~~return~~ ‘Not Played Yet’ else ~~return~~ ‘Match: ’ ~~concatenated with~~ || match ID ~~concatenated with~~ || ‘ Lanes: ’ ~~concatenated with~~ || lanes ~~concatenated with~~ || ‘ Odd Lane Team:’ ~~concatenated with~~ || Teams.team name ~~from the teams table concatenated with~~ || ‘ Even Lane Team: ’ ~~concatenated with~~ || Teams_1.team name ~~from the second copy of the teams table~~ from the tourney matches ~~table~~ inner join~~ed with the~~ teams ~~table~~ on Tourney_Matches.odd lane team ID ~~in the tourney matches table equals~~ = Teams.team ID ~~in the teams table, then~~ inner join~~ed with a second copy of the~~ teams ~~table~~ AS Teams_1 on Tourney_Matches.even lane team ~~in the tourney matches table equals~~ = Teams_1. team ID ~~in the second copy of the teams table, then~~ RIGHT outer join~~ed with the tournaments table~~ on Tourney_Matches.tourney ID ~~in the tourney matches table equals~~ = Tournaments.tourney ID ~~in the tournaments table~~
[依次基于联赛场次表中的单赛道球队 ID 与球队表中的球队 ID 相同将联赛场次表内连接到球队表、基于联赛场次表的双赛道球队 ID 与球队表中的球队 ID 相同内连接到球队表（别名为 Teams_1）、基于联赛 ID 相同右外连接到联赛表。选择联赛 ID、联赛日期、联赛地点和场次详情，其中场次详情是这样确定的：如果来自联赛场次表的场次 ID 为 Null，就返回 Not Played Yet；否则将‘Match: ’、场次 ID、‘ Lanes: ’、比赛赛道、‘ Odd Lane Team: ’、来自球队表第一个副本的球队名、‘ Even Lane Team: ’、来自球队表第二个副本（Teams_1）的球队名拼接起来，并返回拼接得到的结果]

SQL

```
SELECT Tournaments.TourneyID,
  Tournaments.TourneyDate,
  Tournaments.TourneyLocation,
   (CASE WHEN Tourney_Matches.MatchID IS NULL
         THEN 'Not Played Yet'
         ELSE 'Match: ' ||
          CAST(Tourney_Matches.MatchID AS char)
          || ' Lanes: ' || Tourney_Matches.Lanes
          || ' Odd Lane Team: '
          || Teams.TeamName || ' Even Lane Team: '
            || Teams_1.TeamName END) AS Match
FROM ((Tourney_Matches INNER JOIN Teams
  ON Tourney_Matches.OddLaneTeamID =
     Teams.TeamID)
INNER JOIN Teams AS Teams_1
  ON Tourney_Matches.EvenLaneTeamID =
     Teams_1.TeamID)
RIGHT OUTER JOIN Tournaments
  ON Tourney_Matches.TourneyID =
     Tournaments.TourneyID;
```

CH19_All_Tournaments_Any_Matches（63 行）

TourneyID	TourneyDate	TourneyLocation	Match
1	2017-09-04	Red Rooster Lanes	Match: 2 Lanes: 03-04 Odd Lane Team: Terrapins Even Lane Team: Barracudas
1	2017-09-04	Red Rooster Lanes	Match: 3 Lanes: 05-06 Odd Lane Team: Dolphins Even Lane Team: Orcas
1	2017-09-04	Red Rooster Lanes	Match: 4 Lanes: 07-08 Odd Lane Team: Manatees Even Lane Team: Swordfish
1	2017-09-04	Red Rooster Lanes	Match: 1 Lanes: 01-02 Odd Lane Team: Marlins Even Lane Team: Sharks
2	2017-09-11	Thunderbird Lanes	Match: 5 Lanes: 21-22 Odd Lane Team: Terrapins Even Lane Team: Marlins
2	2017-09-11	Thunderbird Lanes	Match: 6 Lanes: 23-24 Odd Lane Team: Barracudas Even Lane Team: Sharks
2	2017-09-11	Thunderbird Lanes	Match: 7 Lanes: 25-26 Odd Lane Team: Dolphins Even Lane Team: Manatees
2	2017-09-11	Thunderbird Lanes	Match: 8 Lanes: 27-28 Odd Lane Team: Swordfish Even Lane Team: Orcas
<< 其他行 >>			

❖ **说明** *在该查询返回的结果末尾，显示的是联赛场次 15~20，它们的场次详情为“Not Played Yet”。*

19.4 小结

本章首先讨论了 CASE 很有用的原因，探索了 CASE 表达式的语法，以及使用 CASE 编写的值表达式中可能使用的查找条件和谓词的语法。接下来阐述了如何使用简单型 CASE 解决问题，并列举了一些示例。然后通过示例详细介绍了查找型 CASE。接着演示了如何在 WHERE 子句中使用 CASE，但指出使用谓词表达式可更清晰地表达你的意图。最后，对于 4 个示例数据库中的每一个，都列举了两个示例，演示了使用简单型 CASE、查找型 CASE 以及在 WHERE 子句中使用 CASE 的其他方式。

下一节列出了你可独自解决的多个问题。

19.5 练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**

  (1) 列出所有的顾客并指出每个顾客是否在 2017 年 12 月的第一周下了订单。

  提示：使用查找型 CASE 以及日期 2107 年 12 月 1 日和 2017 年 12 月 7 日。

  解决方案见 CH19_Customers_Ordered_First-Week_Dec2017（28 行）。

(2) 列出所有的顾客及其所在州的全名。

提示：使用简单型 CASE 并查找 WA、OR、CA、TX 等。

解决方案见 CH19_Customers_State_Names（28 行）。

(3) 显示所有员工及其在 2018 年 2 月 15 日的年龄。

提示：务必使用你使用的数据库系统支持的函数从日期中提取年份、月和日。

解决方案见 CH19_Employee_Age_Feb152018（8 行）。

- **Entertainment Agency 数据库**

(1) 显示所有顾客及其喜欢的音乐风格，但将音乐风格 50's、60's、70's 和 80's 改为 Oldies。

提示：使用简单型 CASE 表达式。

解决方案见 CH19_Customer_Styles_Oldies（36 行）。

(2) 找出演奏爵士乐（Jazz）但不演奏现代音乐（Contemporary）的演唱组合。

提示：在 WHERE 子句中使用查找型 CASE，并从否定的角度思考。

解决方案见 CH19_Entertainers_Jazz_Not_Contemporary（1 行）。

- **School Scheduling 数据库**

(1) 根据婚姻状况编码显示学生的婚姻状况详情。

提示：在 SELECT 子句中使用简单型 CASE。M = Married（已婚）；S = Single（单身）；D = Divorced（离异）；W = Widowed（丧偶）。

解决方案见 CH19_Student_Marital_Status（18 行）。

(2) 计算学生在 2017 年 11 月 5 日的年龄。

提示：务必使用你使用的数据库系统支持的函数从日期中提取年份、月和日。

解决方案见 CH19_Student_Age_Nov15_2017（18 行）。

- **Bowling League 数据库**

(1) 列出所有的投球手并计算它们的平均得分（方法是计算总得分并除以参赛局数，但要注意避免除零错误）。

提示：在使用外连接和 GROUP BY 的查询中使用简单型 CASE。

解决方案见 CH19_Bowler_Averages_Avoid_0_Games（32 行）。

(2) 列出联赛日期、联赛地点、场次、单赛道球队、双赛道球队、局次和获胜球队（对于还未举行的局次，显示 Match not played）。

提示：将联赛表、联赛场次表、球队表、球队表的第二个副本内连接起来；编写一个内使用场次局次表和球队表第三个副本的子查询以获取获胜球队；再将上述两个结果集外连接起来。在 SELECT 子句中，使用查找型 CASE 确定该显示 Match not played 还是获胜球队。

解决方案见 CH19_All_Tourney_Matches（169 行；其中一行表示还未举行的联赛场次 57，它定于 2017 年 11 月 13 日举行）。

第 20 章

使用非连接数据和“驱动”表

“如果你手里只有锤子，就会倾向于将每个问题都看作钉子。”

——亚伯拉罕·马斯洛

本章涵盖如下主题：

- 何谓非连接数据
- 使用非连接数据解决问题
- 使用“驱动”表
- 语句举例
- 小结
- 练习

阅读本章前，务必先好好地睡一觉！另外，在我发出警告的时候，你可能还应确保系好了安全带。本书前面承诺过，要介绍让你跳出习惯思维的概念。本章将介绍可使用非连接据来解决的问题：在 FROM 子句中指定多个表，但不使用 ON 子句指定任何连接条件。我们开始吧。

> ❖ **警告** 本章将大量使用 CASE 表达式。如果不能熟练地使用 CASE 表达式，强烈建议你在阅读本章前详细阅读第 19 章。

20.1 何谓非连接数据

第 7 章开头说过，使用 SQL 解决问题时，大都需要从多个表收集数据。第 8 章演示了如何基于主键和外键匹配来连接多个表，以便从中获取信息。第 9 章演示了如何基于主键和外键匹配从一个表中获取所有行，同时从另一个相关表中获取匹配的信息。在本章中，我将使用多个表，但故意不基于键值匹配；换而言之，我将使用非连接表。

我们来看看创建非连接表的 SQL 语法。先来看看 SELECT 语句的语法，如图 20-1 所示。

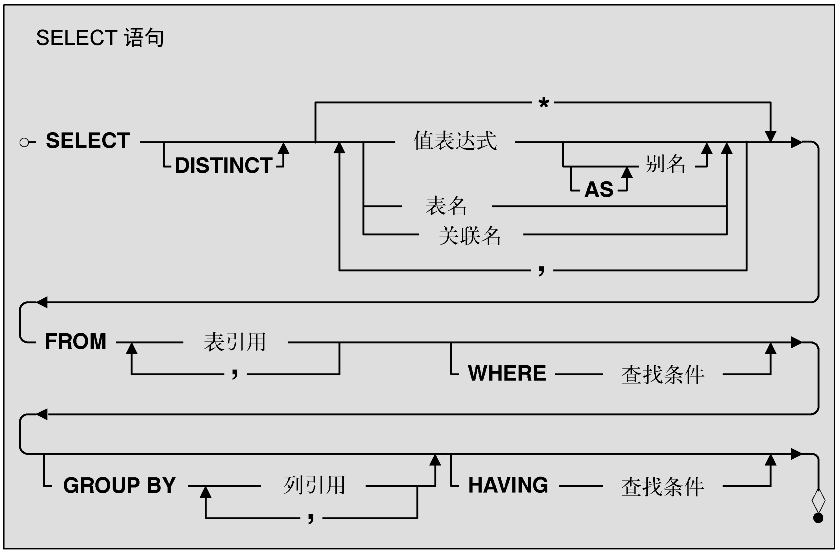


图 20-1　SELECT 语句

要搞明白如何在 FROM 子句中使用非连接表，需要研究表引用（Table Reference）。图 20-2 是完整的表引用语法图。

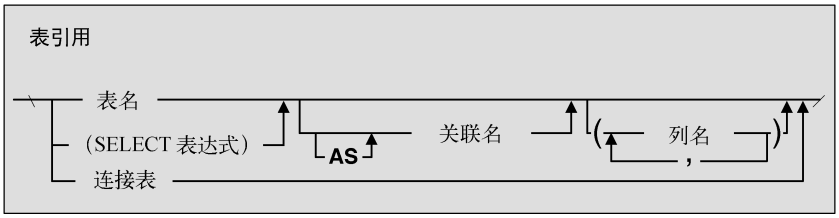


图 20-2　表引用的结构

最后，你需要研究连接表（Joined Table）的语法图。虽然你不会去连接非连接表，但 SQL 标准在连接表定义中指出了如何做，如图 20-3 所示。

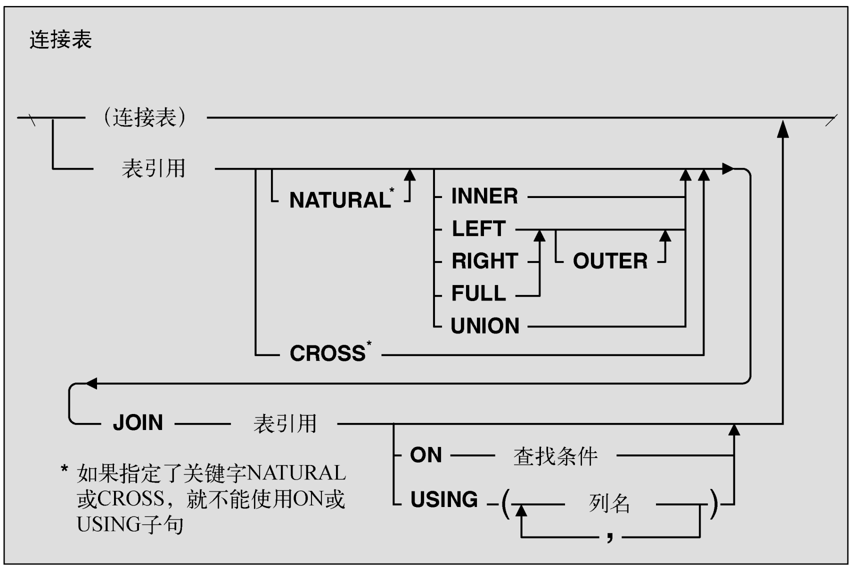


图 20-3　连接表的语法图

要使用非连接表，需要执行 SQL 标准所谓的交叉连接（CROSS JOIN）。在 FROM 子句中交叉连接多个表时，结果是什么呢？结果是所谓的笛卡儿积，即将得到第一个表各行与第二个表各行的组合，总行数为两个表的行数的乘积。我们来看一个简单的示例：

```
SELECT Customers.CustLastName,
  Products.ProductName
FROM Customers CROSS JOIN Products;
```

在示例数据库 Sales Orders 中，有 28 名顾客和 40 种商品，因此结果集将包含 1120（28 × 40）行，类似于下面这样：

CustLastName	ProductName
Viescas	Trek 9000 Mountain Bike
Thompson	Trek 9000 Mountain Bike
Hallmark	Trek 9000 Mountain Bike
Brown	Trek 9000 Mountain Bike
McCrae	Trek 9000 Mountain Bike
Viescas	Trek 9000 Mountain Bike
Sergienko	Trek 9000 Mountain Bike
Patterson	Trek 9000 Mountain Bike
Cencini	Trek 9000 Mountain Bike
Kennedy	Trek 9000 Mountain Bike
<< 其他行 >>	

你可能会问，这怎么就很有用呢？假设要为每位顾客定制一个商品目录，而销售部要求你生成必要的信息，以便能够使用诸如 Dear Mr. Thompson 或 Dear Mrs. Brown 这样的称呼、在封面上打印邮寄标签并列出出售的商品。显然，你可通过 Orders 表和 Order_Details 表将 Customers 表和 Products 表连接起来，但只能获得每位顾客购买过的商品。为了解决这个问题，需要使用非连接表，这将生成笛卡儿积，从而获取需要的信息（顺便说一句，我将列出所有顾客和商品的查询保存到了示例数据库 Sales Orders 中，并将其命名为 CH20_Customer_Catalog）。

❖ **说明** SQL 标准允许你列出各个表并用逗号分隔，以使用非连接表（参见图 20-1 所示的 SELECT 语句的语法图），而且几乎所有的数据库系统都支持这种语法。但前面说过，SQL 标准还定义了关键字 CROSS JOIN，让你能够明确地指出你要获取该关键字左边和右边的表引用的笛卡儿积。

在 Microsoft SQL Server 中，保存只使用了逗号来分隔表名的视图时，你将发现在保存的视图中，用关键字 CROSS JOIN 替换了逗号。在 MySQL 中，保存只使用了逗号的视图时，你将发现在保存的视图中，用关键字 JOIN 替换了逗号（如果没有指定关键字 INNER 或 OUTER 且没有包含 ON 子句，默认为 CROSS）。PostgreSQL 保留逗号，但将 INNER JOIN 替换为 JOIN（没有指定时默认为 INNER）。Microsoft Office Access 不支持关键字 CROSS JOIN，因此创建示例查询时，我只使用最小公分母——逗号语法，但在正文和“整理”中，我依然会使用 CROSS JOIN，让你清楚地知道我要做什么。在示例数据库中的 SQL 语句中，我只使用逗号。

什么情况下该使用交叉连接

判断是否该使用交叉连接并不容易。需要使用交叉连接的查询分为两类。

- ❑ 使用数据库中多个主表（用于存储应用程序描述的所有客体和操作的表）的数据。

本章前面举的例子是列出所有顾客和商品，但也可以是所有经纪人和演唱组合、所有学生和课程、所有可能的球队间比赛（使用 Teams 表的两个副本，但不连接它们）。

- ❑ 使用一个或多个主表中的数据以及一个辅助表（驱动表）中的数据。例如，驱动表可能包含特定时段内的所有日期。

你的数据库中当然包含日期信息，如 Orders 表中的 OrderDate，但如果要考虑特定时段内的每个日期，而不管在这个日期是否有订单，就需要使用驱动（driver）来提供所有的值。你也可以使用驱动表来提供“查找”值，如性别编码到相关单词的转换或数字成绩到字母表示的转换。

20.2　使用非连接数据解决问题

通常，着手使用主表中的数据解决问题时，你先确定所需的数据存储在什么地方，再以有意义的方式将这些表关联起来。所需数据存储在多个表中时，你使用连接将表关联起来，这包括为建立连接所需的中间表，虽然你不需要这些中间表中的数据。

使用非连接数据解决问题时，需要打破这种思维模式，跳出习惯思维，这样才能获得所需的答案。我们再来看一下前面的商品目录问题，但让它更复杂些——将顾客购买过的商品标出来。

> ❖ **说明**　贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”流程。考虑到你现在对这个流程已非常熟悉，为简化流程，我在下面的示例中都将转换和整理合二为一了。

“生成一个清单，其中包含所有顾客的姓名和地址以及我们出售的所有商品，并将顾客购买过的商品标出来。”

基于在第三部分学到的知识，着手处理这个问题时，你首先会研究表间关系。图 20-4 显示了标准做法：通过 Orders 表和 Order_Details 表将 Customers 表关联到 Products 表。

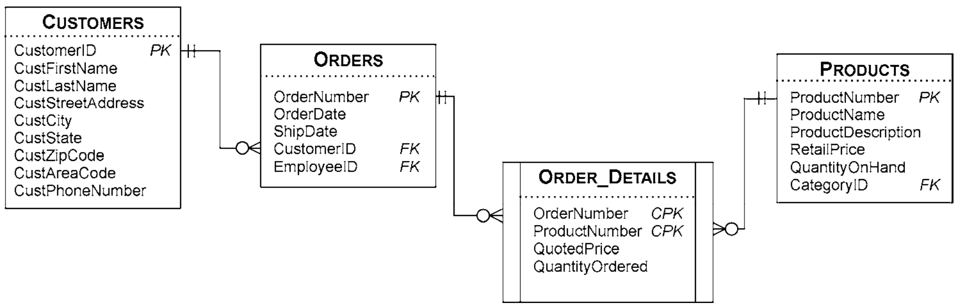


图 20-4　将 Customers 表关联到 Products 表的常规方式

别忘了，我需要获取所有的顾客（包括什么都没买的顾客）和所有商品（包括没有顾客购买的商品）。如果你积极思考了，可能想出了使用全外连接（参见第 9 章）的方案。这种想法没错，这是解决这个问题的方式之一。别忘了，并非所有数据库系统都支持全外连接，因此你可能无法使用这种解决方案。你也可编写一个查询（视图），将 Customers 表左外连接到 Orders 表和 Order_Details 表，再在另一个查询中将该查询的结果右外连接到 Products 表。在这样得到的结果中，如果 Order_Details 表的键列的值不为 Null，就说明顾客购买过相应的商品（还记得 CASE 吗）。

但本章的主题是使用非连接数据解决问题，因此我们正面“应敌”，这样来解决这个问题：交叉连接 Customers 表和 Products 表，并在 SELECT 子句中使用子查询确定当前顾客是否购买过当前商品。出于好玩，我们也查找每种商品的类别描述。下面就来这样做。

转换/整理	Select cust~~omer~~ first name, cust~~omer~~ last name, cust~~omer~~ street address, cust~~omer~~ city, cust~~omer~~ state, cust~~omer~~ zip code, category description, product number, product name, retail price, ~~and~~ (CASE when ~~the~~ customer ID ~~is~~ in ~~the~~ (selecti~~on of~~ customer ID from ~~the~~ orders ~~table~~ inner join~~ed with the~~ order details ~~table~~ on Orders.order number ~~in the orders table equals~~ = Order_Details.order number ~~in the order details table~~ where ~~the~~ Products.product number ~~in the products table equals the~~ = Order_Details.product number) ~~in the order details table~~ then ~~display~~ ‘You purchased this!’~~,~~ else ‘ ’ END) ~~display a blan~~k from ~~the~~ customers ~~table and the~~ CROSS JOIN categories ~~table~~ inner join~~ed with the~~ products ~~table~~ on Categories.category ID ~~in the categories table equals~~ = Products.category ID ~~in the products table sorted~~ ORDER BY customer ID, category description, ~~and~~

	product number（基于类别 ID 相同将商品表内连接到类别表，再交叉连接到顾客表；选择顾客名、顾客姓、顾客街道地址、顾客城市、顾客州、顾客邮政编码、类别描述、商品编号、商品名、零售价和是否购买过。是否购买过是这样确定的：基于订单号相同将订单表内连接到订单详情表，从商品编号与当前商品的编号相同的行中选择顾客 ID，生成一个列表；如果当前顾客的 ID 在这个列表中，就返回 You purchased this!，否则返回‘ ’）
SQL	见下方代码

```
SELECT Customers.CustomerID, Customers.
CustFirstName,
  Customers.CustLastName,
  Customers.CustStreetAddress,
  Customers.CustCity, Customers.CustState,
  Customers.CustZipCode,
  Categories.CategoryDescription,
  Products.ProductNumber, Products.ProductName,
  Products.RetailPrice,
    (CASE WHEN Customers.CustomerID IN
      (SELECT Orders.CustomerID
       FROM ORDERS INNER JOIN Order_Details
       ON Orders.OrderNumber =
          Order_Details.OrderNumber
       WHERE Order_Details.ProductNumber =
             Products.ProductNumber)
     THEN 'You purchased this! '
     ELSE ' ' END) AS ProductOrdered
FROM Customers, Categories INNER JOIN Products
       ON Categories.CategoryID = Products.CategoryID
ORDER BY Customers.CustomerID,
    Categories.CategoryDescription,
    Products.ProductNumber;
```

诚然，在 FROM 子句中，将 Categories 表内连接到了 Products 表，但关键部分是到 Customers 表的交叉连接。这个查询保存到了示例数据库 Sales Orders 中，名为 CH20_Customer_All_Products_PurchasedStatus，按照预期，它返回 1120 行。

> ❖ **说明** 第 19 章说过，Microsoft Office Access 不支持 CASE 表达式。在 Access 数据库中的查询示例中，我使用的是功能与 CASE 表达式类似的内置函数 IIf。

20.3 使用“驱动”表解决问题

现在来看需要这样解决的问题：建立一个或多个包含一系列值的表，将它们交叉连接到数据库中的其他表，以获得答案。我将这种表称为驱动（driver）表，因为结果是在其内容的驱动下得到的［如果你还购买了我与好友 Doug Steele 和 Ben Clothier 合著的 *Effective SQL*，将发现该书称之为记录（tally）表，但说的是一码事］。最常见的一种驱动表包含日期、星期或月份列表，你可将其交叉连接到数据表，以列出所有的日期、星期或月份，以及在这些时间发生的事件。

驱动表的另一种用途是定义对特定范围内的值进行的分类，这包括给数字成绩指定相应的字母表示、根据教员的评分情况评定等级、根据平均得分评价投球手、根据价格将商品归类、根据消费情况将顾客归类。

驱动表的一种极具创意的用法是“透视”数据，让显示的结果更像是电子表格。一个常见的例子是，显示商品或顾客每月的销量或消费额。

20.3.1 建立驱动表

SQL 标准定义了 WITH RECURSIVE，让你能够在循环中多次执行指定的 SQL 查询，这在你需要将特定范围内连续的日期加载到驱动表中时很有用，但只有为数不多的数据库系统支持这种功能。为了加载我的大型驱动表，我在 Microsoft Office Access 中使用 Visual Basic 来执行必要的递归，将数百行加载到一个日期范围表中（只要深挖 Microsoft Office Access 版示例数据库，就能找到我使用的一些代码）。

如果驱动表只是简单的范围到值的映射，手动加载数据也很容易。例如，下面是我在示例数据库 School Scheduling 的 ztblLetterGrades 表中输入的一系列值：

LetterGrade	LowGradePoint	HighGradePoint
A	93	96.99
A–	90	92.99
A+	97	120
B	83	86.99
B–	80	82.99
B+	87	89.99
C	73	76.99
C–	70	72.99
C+	77	79.99
D	63	66.99
D–	60	62.99
D+	67	69.99
F	0	59.99

这些值你应该觉得眼熟，因为它们就是前一章的查询 CH19_Students_Classes_Letter_Grades 中使用的范围列表（顺便说一句，我给所有驱动表命名时，都以 ztbl 打头，旨在将它们与数据库中的主数据表分开）。建立这种表的一个好处是，需要时可轻松地修改范围值，同时无须在每个依靠范围来获得答案的查询中使用 CASE。

前面说过，驱动表的一种极具创意的用法是“透视”结果，使其看起来更像电子表格。很多数据库系统提供了透视数据的非标准方式，但我将演示如何使用标准 SQL 和驱动表来透视数据。在示例数据库 Sales Orders 中，我为此创建了驱动表 ztblMonths，下面显示了其部分内容。

MonthYear	YearNumber	MonthNumber	MonthStart	MonthEnd
January 2017	2017	1	1/1/2017	1/31/2017
February 2017	2017	2	2/1/2017	2/29/2017
March 2017	2017	3	3/1/2017	3/31/2017
April 2017	2017	4	4/1/2017	4/30/2017
May 2017	2017	5	5/1/2017	5/31/2017
June 2017	2017	6	6/1/2017	6/30/2017
July 2017	2017	7	7/1/2017	7/31/2017
August 2017	2017	8	8/1/2017	8/31/2017
		<< 其他行 >>		

另外一些列……

January	February	March	April	May	June
1	0	0	0	0	0
0	1	0	0	0	0
0	0	1	0	0	0
0	0	0	1	0	0
0	0	0	0	1	0
0	0	0	0	0	1
0	0	0	0	0	0
0	0	0	0	0	0
		<< 其他行 >>			

另外一些列……

July	August	September	October	November	December
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	0
<< 其他行 >>					

看起来有点怪？这里的小秘密是，你将使用 WHERE 子句来找出这个驱动表中与下单日期匹配的行，再将销售总额乘以特定列的值来得到一个月的销售总额。如果订单是 2017 年 1 月份的，则在匹配的行中，只有 January 列的值为 1，因此将把 1 乘以订购数量再乘以零售价。与表示其他月份的列中的值相乘对结果没有影响，因为零与任何值的乘积都为零。你也可以这样认为：对于查询遇到的每个值，都有一系列由 1 和 0 定义的水平“桶”，这些桶被用来计算最重要显示的值。日期与驱动表中某行定义的范围匹配时，1 指出了要将值放到哪个水平桶中。因此，当日期落在 2017 年 1 月时，最终的值将放到 January 列中，而最终值的计算方法是将该列的值与计算销售总额的表达式相乘。

20.3.2　使用驱动表

下面使用前一小节说的两个驱动表来解决问题。首先，根据学生的数字成绩显示字母表示。前一章使用 CASE 解决了这个问题，这里将使用驱动表来解决。

“列出所有的学生及其注册的课程以及每门课程的成绩和字母表示。”

转换/整理

Select Students.student ID ~~from the students table~~, Students. stud~~ent~~ first name ~~from the students table~~, Students.stud~~ent~~ last name ~~from the students table~~, Classes.class ID ~~from the classes table~~, Classes.start date ~~from the classes table~~, Subjects.subject code ~~from the subjects table~~, Subjects.subject name ~~from the subjects table~~, Student_Schedules.grade ~~from the student_schedules table~~, ~~and~~ ztblLetterGrades.letter grade ~~from the letter grades driver table~~ from ztblLetterGrades ~~the letter grades driver table and~~ CROSS JOIN ~~the~~ students ~~table~~ inner join~~ed with the~~ student schedules ~~table~~ on Students.student ID ~~in the students table equals~~ = Student_Schedules.student ID ~~in the student schedules table, then~~ inner join~~ed with the~~ classes ~~table~~ on Student_Schedules.class ID ~~in the student schedules table equals~~ = Classes.class ID ~~in the classes table, then~~ inner join~~ed with the~~ subjects ~~table~~ on Classes.subject ID ~~in the classes table equals~~ = Subjects.subject ID ~~in the subjects table, then~~ inner join~~ed with the~~ student class status ~~table~~ on Student_Schedules.class status ~~in the student schedules table~~ equals = Student_Class_Status.class status ~~in the student class status table~~ where Student_Class_Status.class status description ~~in the student class status table equals~~ = ‘Completed’ and Student_Schedules.grade ~~in the student schedules table is~~ between ztblLetterGrades.low grade point ~~in the letter grades driver table~~ and ztblLetterGrades.high grade point in ~~the letter grades driver table~~（依次基于学生 ID 相同将学生表内连接到学生选课表、基于课程 ID 相同内连接到课程表、基于科目 ID 相同内连接到科目表、基于课程状态相同内连接到学生课程状态表，生成一个结果集；将字母成绩表交叉连接到上述结果集；从课程状态为 Completed 且成绩位于字母成绩表中最高成绩和最低成绩之间的行中，选择学生 ID、学生名、学生姓、课程 ID、开始日期、科目编码、科目名、成绩和字母成绩）

SQL

```
SELECT Students.StudentID, Students.StudFirstName,
  Students.StudLastName, Classes.ClassID,
  Classes.StartDate,
  Subjects.SubjectCode, Subjects.SubjectName,
  Student_Schedules.Grade,
  ztblLetterGrades.LetterGrade
FROM ztblLetterGrades, (((Students
INNER JOIN Student_Schedules
  ON Students.StudentID =
     Student_Schedules.StudentID)
INNER JOIN Classes
  ON Student_Schedules.ClassID = Classes.ClassID)
```

```
INNER JOIN Subjects
  ON Classes.SubjectID = Subjects.SubjectID)
INNER JOIN Student_Class_Status
  ON Student_Schedules.ClassStatus =
     Student_Class_Status.ClassStatus
WHERE
  (Student_Class_Status.ClassStatusDescription =
      'Completed')
  AND (Student_Schedules.Grade Between
     ztblLetterGrades.LowGradePoint
     AND ztblLetterGrades.HighGradePoint);
```

这个查询保存到了示例数据库 School Scheduling 中，名为 CH20_Students_Classes_Letter_Grades。与第 19 章的查询 CH19_Student_Classes_Letter_Grades 一样，这个查询也返回 68 行。

下面来使用第二个驱动表。

“列出每种商品及其各月的总销售额。”

转换/整理

Select Products.product name ~~from the products table~~, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths.January ~~from the months driver table~~) as January, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths.February ~~from the months driver table~~) as February, ~~the~~ sum ~~of~~ (Order_Details. quoted price ~~from the order details table times~~ * Order_ Details.quantity ordered ~~from the order details table times~~ * ztblMonths.March ~~from the months driver table~~) as March, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths.April ~~from the months driver table~~) as April, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths.May ~~from the months driver table~~) as May, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths. June ~~from the months driver table~~) as June, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths.July ~~from the months driver table~~) as July, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths.August ~~from the months driver table~~) as August, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths.September ~~from the months driver table~~) as September, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_ Details.quantity ordered ~~from the order details table times~~ * ztblMonths.October ~~from the months driver table~~) as October, ~~the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths.November ~~from the months driver table~~) as November, ~~and the~~ sum ~~of~~ (Order_Details.quoted price ~~from the order details table times~~ * Order_Details.quantity ordered ~~from the order details table times~~ * ztblMonths.December ~~from the months driver table~~) as December from ztblMonths ~~the months driver table~~ CROSS JOIN ~~and the~~ products ~~table then~~ inner join~~ed with the~~ order details ~~table~~ on Products.product number ~~in the products table equals~~ = Order_Details.product number ~~in the order details table then~~ inner join~~ed with the~~ orders ~~table~~ on Orders.order number ~~in the orders table equals~~ = Order_Details.order number ~~in the order details table~~ where Orders.order date in the orders table is between ztblMonths.month start ~~in the months driver table~~ and ztblMonths.month end ~~in the months driver table~~ group~~ed~~ by Products.product name ~~in the products table~~［依次基于商品编号相同将商品表内连接到订单详情表、基于订单号相同内连接到订单表，生成一个结果集；将月份驱动表交叉连接到上述结果集，选择订单日期在月份驱动表的月份开始日期和终止日期之间的行，并按商品名分组；从每个分组中，选择商品名、报价与订购数量和月份驱动表中 January 列值的乘积总和值（作为 January）、报价与订购数量和月份驱动表中 February 列值的乘积总和值（作为 February）、报价与订购数量和月份驱动表中 March 列值的乘积总和值（作为 March）、报价与订购数量和月份驱动表中 April 列值的乘积总和值（作为 April）、报价与订购数量和月份驱动表中 May 列值的乘积总和值（作为 May）、报价与订购数量和月份驱动表中 June 列值的乘积总和值（作为 June）、报价与订购数量和月份驱动表中 July 列值的乘积总和值（作为 July）、报价与订购数量和月份驱动表中 August 列值的乘积总和值（作为 August）、报价与订购数量和月份驱动表中 September 列值的乘积总和值（作为 September）、报价与订购数量和月份驱动表中 October 列值的乘积总和值（作为 October）、报价与订购数量和月份驱动表中 November 列值的乘积总和值（作为 November）、报价与订购数量和月份驱动表中 December 列值的乘积总和值（作为 December）］

SQL

```
SpELECT Products.ProductName,
  SUM(Order_Details.QuotedPrice *
    Order_Details.QuantityOrdered *
    ztblMonths.January)
    AS January,
  SUM(Order_Details.QuotedPrice *
    Order_Details.QuantityOrdered *
    ztblMonths.February)
```

```
        AS February,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.March)
        AS March,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.April)
        AS April,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.May)
        AS May,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.June)
        AS June,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.July)
        AS July,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.August)
        AS August,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.September)
        AS September,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.October)
        AS October,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.November)
        AS November,
      SUM(Order_Details.QuotedPrice *
        Order_Details.QuantityOrdered *
        ztblMonths.December)
        AS December
    FROM ztblMonths, (Products
    INNER JOIN Order_Details
      ON Products.ProductNumber =
         Order_Details.ProductNumber)
    INNER JOIN Orders
      ON Orders.OrderNumber = Order_Details.OrderNumber
    WHERE Orders.OrderDate BETWEEN ztblMonths.MonthStart
       AND ztblMonths.MonthEnd
    GROUP BY Products.ProductName;
```

当你进行限制，只返回驱动表中月份开始日期和终止日期与订单日期匹配的行时，奇迹便发生了。如果订单日期落在 1 月份，只有 January 列的值为 1。这将把相应行的销售总额放到正确的桶中，进而得以汇总来获得透视结果。这个查询保存到了示例数据库 Sales Orders 中，名为 CH20_Product_Sales_Month_Pivot。这个查询返回 38 行，但商品表中有 40 中商品。之所以少了两行，是因为有两种商品没有人购买（要知道是哪两种商品，请参阅 CH09_Products_Never_Ordered）。

20.4 语句举例

至此，你知道了如何使用交叉连接和驱动表来编写查询，还知道了使用这些方法可解决的一些问题类型，下面来看一些示例，它们都交叉连接了两个数据表（或将数据表交叉连接到驱动表）。这些示例都来自示例数据库，演示了如何跳出习惯思维，使用这些方法来解决问题。

> ❖ **说明**　本书前言提醒过，在你使用的数据库系统中，行的排列顺序不一定与本书示例中显示的相同——除非使用 ORDER BY 子句。即便指定了排序方式，在数据库系统返回的结果中，对于 ORDER BY 子句中未包含的列，根据它们的排列顺序也可能不同，这是因为优化方法因数据库系统而异。
>
> 如果你在 Microsoft SQL Server 中运行示例，则从视图中选择行时，将不会遵循其中的 ORDER BY 子句指定的排列顺序。要遵循 ORDER BY 子句指定的排列顺序，必须打开视图的设计，并在显示界面中执行它。
>
> 另外，当你使用 GROUP BY 子句时，数据库系统返回的结果集看起来像是根据该子句中指定的列进行了排序。这是因为有些优化器首先会在内部对数据排序，以提高 GROUP BY 的处理速度。别忘了，如果要按特定的顺序排列，必须使用 ORDER BY 子句。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库中，名称以 CH20 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明**　别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录 B。这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。别忘了，对于包含 ORDER BY 子句的 SQL Server 视图，必须现在“设计”模式下打开再执行它，结果才会按指定顺序排列；如果你使用 SELECT * 获取视图内容，SQL Server 将不会按 ORDER BY 子句指定的顺序排列结果集。

20.4.1　非连接表使用示例

下面的示例演示了使用非连接数据可以解决的问题，它们都交叉连接了两个数据表。

- **Sales Orders 数据库**

“成对地列出居住在同一个州的员工和顾客，并指出顾客是否通过相应的员工下过订单。”

转换/整理

Select emp~~loyee~~ first name, emp~~loyee~~ last name, cust~~omer~~ first name, cust~~omer~~ last name, cust~~omer~~ area code, cust~~omer~~ phone number, ~~and~~ (CASE when ~~the~~ customer ID ~~in the customers table is~~ in ~~the~~ (selec~~tion of~~ Orders. customer ID ~~from the orders table~~ where Orders.employee ID ~~in the orders table equals~~ = Employees.employee ID) ~~in the employees table~~ then ~~display~~ ‘Ordered from you.’ else ~~display~~ ‘ ’ END) from employees ~~and~~ CROSS JOIN customers where ~~the~~ Employees.emp~~loyee~~ state ~~in the employees table equals~~ = Customer.cust~~omer~~ state ~~in the customers table~~（将员工表交叉连接到顾客表，从顾客所在州与员工所在州相同的行中，选择员工名、员工姓、顾客名、顾客姓、顾客区号、顾客电话号码和是否通过相应的员工下过订单。是否通过相应的员工下过订单是这样确定的：在订单表中，从员工 ID 与当前员工的 ID 相同的行中选择顾客 ID，生成一个列表；如果当前顾客的 ID 包含在这个列表中，就显示“ Ordered from you.”，否则显示‘ ’）

SQL

```
SELECT Employees.EmpFirstName,
  Employees.EmpLastName,
  Customers.CustFirstName,
  Customers.CustLastName,
  Customers.CustAreaCode,
  Customers.CustPhoneNumber,
  (CASE WHEN Customers.CustomerID IN
           (SELECT Orders.CustomerID
            FROM Orders
            WHERE Orders.EmployeeID =
                  Employees.EmployeeID)
        THEN 'Ordered from you. '
        ELSE ' ' END) AS CustStatus
FROM Employees, Customers
WHERE Employees.EmpState = Customers.CustState;
```

CH20_Employees_Same_State_Customers（83 行）

Emp FirstName	Emp LastName	Cust FirstName	Cust LastName	Cust AreaCode	Cust PhoneNumber	Cust Status
Ann	Patterson	William	Thompson	425	555-2681	Ordered from you.
Ann	Patterson	Gary	Hallmark	253	555-2676	Ordered from you.
Ann	Patterson	Dean	McCrae	425	555-2506	Ordered from you.
Ann	Patterson	John	Viescas	425	555-2511	Ordered from you.
Ann	Patterson	Andrew	Cencini	206	555-2601	Ordered from you.
Ann	Patterson	Liz	Keyser	425	555-2556	Ordered from you.
Ann	Patterson	Julia	Schnebly	206	555-9936	Ordered from you.
Ann	Patterson	Suzanne	Viescas	425	555-2686	Ordered from you.
Ann	Patterson	Jeffrey	Tirekicker	425	555-9999	
Ann	Patterson	Joyce	Bonnicksen	425	555-2726	Ordered from you.
<< 其他行 >>						

❖ **说明** 如果你眼光敏锐，可能知道原本可通过基于 EmpState = CustState 将 Employees 表内连接到 Customers 表来解决这个问题。本书前面说过很多次，几乎所有问题都有多种解决方式。现在你知道如何使用交叉连接来解决这个问题了。

- **Entertainment Agency 数据库**

“列出有顾客喜欢的每种音乐风格，并指出每种音乐风格是多少顾客最喜欢的、多少顾客第二喜欢的、多少顾客第三喜欢的。”

❖ **说明** 这个问题有点棘手，因为你首先需要“透视”每位顾客最喜欢、第二喜欢和第三喜欢的音乐风格（PreferenceSeq 列指出了这一点），再进行计数。你可使用驱动表来帮助执行透视，但由于只需将 3 个不同的值透视到列中，因此使用 CASE 来透视也很容易。

转换/整理

Select Musical_Styles.style ID ~~from the musical styles table~~, Musical_Styles.style name ~~from the musical styles table~~, ~~the~~ count ~~of~~ (RankedPeferences.first style) ~~from the ranked preferences query~~ as first preference, ~~the~~ count ~~of~~ (RankedPreferences.second style) ~~from the ranked preferences query~~ as second preference, ~~and the~~ count ~~of~~ (RankedPreferences.third style) ~~from the ranked preferences query~~ as third preference from ~~the~~ musical styles ~~table and~~ CROSS JOIN ~~the~~ (selec~~tion of~~ (CASE when Musical_Preferences. preference sequence ~~in the musical preferences table is~~ = 1 then ~~return the~~ Musical_Preferences.style ID ~~from the musical preferences table~~ else ~~return~~ Null END) as first style, (CASE when Musical_Preferences.preference sequence ~~in the musical preferences table is~~ = 2 then ~~return the~~ Musical_Preferences.style ID ~~from the musical preferences table~~ else ~~return~~ Null END) as second style, (CASE when Musical_Preferences.preference sequence ~~in the musical preferences table is~~ = 3 then ~~return the~~ Musical_Preferences. style ID ~~from the musical preferences table~~ else ~~return~~ Null END) as third style from ~~the~~ musical preferences ~~table~~) as ranked preferences where Musical_Styles.style ID ~~in the musical styles table equals~~ = RankedPreferences.first style ~~in the ranked preferences query~~ or Musical_Styles.style ID ~~in the musical styles table equals~~ = RankedPreferences.second style ~~in the ranked preferences query~~ or Musical_Styles.style ID ~~in the musical styles table equals~~ = RankedPreferences.third style ~~in the ranked preferences query~~ grouped by style ID, ~~and~~ style name having ~~the~~ count ~~of~~ (first style) > ~~greater than~~ 0 or ~~the~~ count ~~of~~ (second style) > ~~greater than~~ 0 or ~~the~~ count ~~of~~ (third style) > ~~greater than~~ 0 order~~ed~~ by first preference descending, second preference descending, third preference descending, ~~and~~ style ID［从顾客喜好表中选择 first style、second style 和 third style，生成一个结果集，并将其命名为 RankedPreferences。具体的方法如下：如果喜好顺序为 1，就将 first style 列设置为相

应的风格 ID，并将 second style 和 third style 列都设置为 Null；如果喜好顺序为 2，就将 second style 列设置为相应的风格 ID，并将 first style 和 third style 列都设置为 Null；如果喜好顺序为 3，就将 third style 列设置为相应的风格 ID，并将 first style 和 second style 列都设置为 Null；否则就将 first style、second style 和 third style 列都设置为 Null。将音乐风格表交叉连接到结果集 RankedPreferences，选择来自音乐风格表的风格 ID 与来自结果集的 first style、second style 或 third style 相同的行，并按风格 ID 和风格名分组。对于每个分组，计算 first style 列不为 Null 的行数、second style 列不为 Null 的行数以及 third style 列不为 Null 的行数。如果这三个行数至少有一个大于零，就从该分组中选择风格 ID、风格名、first style 列不为 Null 的行数（作为 first preference）、second style 列不为 Null 的行数（作为 second preference）以及 third style 列不为 Null 的行数（作为 third preference）]

SQL

```sql
SELECT Musical_Styles.StyleID,
  Musical_Styles.StyleName,
  COUNT(RankedPreferences.FirstStyle)
    AS FirstPreference,
  COUNT(RankedPreferences.SecondStyle)
    AS SecondPreference,
  COUNT(RankedPreferences.ThirdStyle)
    AS ThirdPreference
FROM Musical_Styles,
 (SELECT (CASE WHEN
    Musical_Preferences.PreferenceSeq = 1
               THEN Musical_Preferences.StyleID
               ELSE Null END) As FirstStyle,
         (CASE WHEN
    Musical_Preferences.PreferenceSeq = 2
               THEN Musical_Preferences.StyleID
               ELSE Null END) As SecondStyle,
         (CASE WHEN
    Musical_Preferences.PreferenceSeq = 3
               THEN Musical_Preferences.StyleID
               ELSE Null END) AS ThirdStyle
  FROM Musical_Preferences) AS RankedPreferences
WHERE Musical_Styles.StyleID =
          RankedPreferences.FirstStyle
   OR Musical_Styles.StyleID =
          RankedPreferences.SecondStyle
   OR Musical_Styles.StyleID =
          RankedPreferences.ThirdStyle
GROUP BY StyleID, StyleName
HAVING COUNT(FirstStyle) > 0
      OR COUNT(SecondStyle) > 0
      OR COUNT(ThirdStyle) > 0
ORDER BY FirstPreference DESC,
         SecondPreference DESC,
         ThirdPreference DESC, StyleID;
```

CH20_Customer_Style_Preference_Rankings（20 行）

StyleID	StyleName	FirstPreference	SecondPreference	ThirdPreference
21	Standards	2	2	0
15	Jazz	2	1	0
19	Rhythm and Blues	2	0	1
22	Top 40 Hits	2	0	0
10	Contemporary	1	2	0
8	Classic Rock & Roll	1	1	0
20	Show Tunes	1	1	0
3	60’s Music	1	0	0
11	Country Rock	1	0	0
14	Chamber Music	1	0	0
23	Variety	1	0	0
		<< 其他行 >>		

请注意，虽然数据库中定义了 25 种音乐风格，但这个查询只返回了 20 行。那些缺席的音乐风格不是任何顾客最喜欢、第二喜欢或第三喜欢的。

- **School Scheduling 数据库**

“列出所有修完了英语课的学生，并根据成绩计算他们所处的五分位区间。”

五分位法将一组分成 5 个相等的区间。用于学生排名时，每个五分位区间包含 20%的学生：排在前 20%的位于第一个五分位区间，接下来的 20%位于第二个五分位区间，以此类推。要解决这个问题，你需要交叉连接两个查询。

(1) 一个查询计算修完了英语课的每个学生的排名。要计算一个学生的排名，可计算成绩不比他低的学生数量。成绩最高的学生为第一名，第二高为第二名，以此类推。

(2) 一个查询计算修完了英语课的学生总数。你可将这个数字乘以 0.2、0.4、0.6 和 0.8 来得到五分位数。第一个查询计算得到的排名小于等于学生总数 0.2 倍的，在第一个五分位区间。

转换/整理

Select S1.subject ID ~~from the first query~~, S1.stud~~ent~~ first name ~~from the first query~~, S1.stud~~ent~~ last name ~~from the first query~~, S1.class status ~~from the first query~~, S1.grade ~~from the first query~~, S1.category ID ~~from the first query~~, S1.subject name ~~from the first query~~, S1.rank in category ~~from the first query~~, StudCount.num~~ber of~~ students ~~from the student count query~~, ~~and~~ (CASE when ~~the~~ rank in category <= ~~is less than or equal to~~ 0.2 * ~~times the~~ num~~ber of~~ students then ~~return~~ 'First' when the rank in category <= ~~is less than or equal to~~ 0.4 * ~~times the~~ num~~ber of~~ students then ~~return~~ 'Second' when ~~the~~ rank in category <= ~~is less than or equal to~~ 0.6 * ~~times the~~ num~~ber of~~ students then ~~return~~ 'Third' when ~~the~~ rank in category <= ~~is less than or equal to~~ 0.8 * ~~times the~~ num~~ber of~~ students then ~~return~~ 'Fourth' else ~~return~~ 'Fifth' END) as ~~the~~ quintile from ~~the~~ (selec~~tion of~~ Subjects.subject ID ~~in the subjects table~~, Students.stud~~ent~~ first name ~~in the students table~~, Students. stud~~ent~~ last name ~~in the students table~~, Student_Schedules.class status ~~in the student schedules table~~, Student_Schedules.grade ~~in the student schedules table~~, Subjects.category ID ~~in the subjects table~~, Subjects.subject name ~~in the subjects table~~, ~~and the~~ (selec~~tion of the~~ count(*) ~~of all rows~~ from ~~the~~ classes ~~table~~ inner join~~ed with the~~ student schedules ~~table~~ AS SS2 on Classes.class ID ~~in the classes table~~ = ~~equals~~ SS2.class ID ~~in the student schedules table, then~~ inner join~~ed with the~~ subjects ~~table~~ AS S3 on S3.subject ID ~~in the subjects table~~ = ~~equals~~ Classes.subject ID ~~in the classes table~~ where S3.category ID ~~in the subjects table~~ = ~~equals~~ 'ENG' and SS2.grade ~~in the student schedules table~~ >= ~~is greater than or equal to~~ Student_Schedules.grade ~~in the student schedules table~~) as rank in category) from ~~the~~ subjects ~~table~~ inner join~~ed with the~~ classes ~~table~~ on Subjects.subject ID ~~in the subjects table~~ = ~~equals~~ Classes.subject ID ~~in the classes table, then~~ inner join~~ed with the~~ student schedules ~~table~~ on Student_Schedules.class ID ~~in the student schedules table~~ = ~~equals~~ Classes.class ID ~~in the classes table, then~~ inner join~~ed with the~~ students ~~table~~ on Students.student ID ~~in the students table~~ = ~~equals~~ Student_Schedules.student ID ~~in the student schedules table~~ where Student_Schedules.class status ~~in the student schedules table~~ = ~~equals~~ 2 and Subjects.category ID ~~in the subjects table~~ = ~~equals~~ 'ENG') AS S1 CROSS JOIN ~~and the~~ (selec~~tion of the~~ count(*) ~~of all rows~~ as num~~ber of~~ students from ~~the~~ classes ~~table~~ AS C2 inner join~~ed with the~~ student schedules ~~table~~ AS SS3 on C2.class id ~~in the classes table~~ = ~~equals~~ SS3.class ID ~~in the student schedules table, then~~ inner join~~ed with the~~ subjects ~~table~~ AS S2 on S2.subject ID ~~in the subjects table~~ = ~~equals~~ C2.subject ID ~~in the classes table~~ where SS3.class status ~~in the student schedules table~~ = ~~equals~~ 2 and S2.category ID ~~in the subjects table~~ = ~~equals~~ 'ENG') As stud~~ent~~ count order~~ed~~ by S1.grade ~~in the first query~~ desc~~ending~~ [依次基于科目 ID 相同将科目表内连接到课程表、基于课程 ID 相同内连接到学生选课表、基于学生 ID 相同内连接到学生表，选择课程状态为 2 且类别 ID 为 ENG 的行，生成一个结果集。从这个结果集中选择来自科目表的科目 ID、来自学生表的名和姓、来自学生选课表的课程状态和成绩、来自科目表的类别 ID 和科目名称以及学生排名，生成一个结果集并将其命名为 S1。其中学生排名是这样计算得到的：依次基于课程 ID 相同将课程表内连接到学生选课表（别名为 SS2）、基于科目 ID 相同内连接到科目表（别名为 S3），选择来自 S3 的类别 ID 为 ENG 且来自 SS2 的成绩不低于当前学生的成绩的行，再计算总行数。依次基于课程 ID 相同将课程表（别名为 C2）内连接到学生选课表（别名为 SS3）、基于科目 ID 相同内连接到科目表（别名为 S2），选择来自 SS3 的课程状态为 2 且来自 S2 的类别 ID 为 ENG 的行，计算总行数并将其作为“学生数”，再将这个结果集命名为“学生计数”。将结果集 S1 交叉连接到结果集“学生计数”，选择来自 S1 的科目 ID、学生姓名、学生名、课程状态、成绩、类别 ID、科目名、学生排名、来自数据集“学生计数”的“学生数”以及五分位区间，并按来自 S1 的成绩排序。五分位区间是这样确定的：如果学生排名小于等于“学生数”的 0.2 倍，就返回 First；如果学生排名小于等于“学生数”的 0.4 倍，就返回 Second；如果学生排名小于等于“学生数”的 0.6 倍，就返回 Third；如果学生排名小于等于“学生数”的 0.8 倍，就返回 Fourth；否则返回 Fifth]

SQL

```
SELECT S1.SubjectID, S1.StudFirstName,
  S1.StudLastName,
  S1.ClassStatus, S1.Grade, S1.CategoryID,
  S1.SubjectName,
  S1.RankInCategory, StudCount.NumStudents,
  (CASE WHEN RankInCategory <= 0.2 * NumStudents
        THEN 'First'
        WHEN RankInCategory <= 0.4 * NumStudents
        THEN 'Second'
        WHEN RankInCategory <= 0.6 * NumStudents
        THEN 'Third'
        WHEN RankInCategory <= 0.8 * NumStudents
```

```
                THEN 'Fourth'
                ELSE 'Fifth' END) AS Quintile
        FROM
          (SELECT Subjects.SubjectID,
             Students.StudFirstName,
             Students.StudLastName,
             Student_Schedules.ClassStatus,
             Student_Schedules.Grade, Subjects.CategoryID,
             Subjects.SubjectName,
            (SELECT Count(*)
             FROM (Classes
             INNER JOIN Student_Schedules AS SS2
                  ON Classes.ClassID = SS2.ClassID)
             INNER JOIN Subjects As S3
                  ON S3.SubjectID = Classes.SubjectID
             WHERE S3.CategoryID = 'ENG'
                  AND SS2.Grade >=
                    Student_Schedules.Grade)
                      AS RankInCategory
          FROM ((Subjects INNER JOIN Classes
              ON Subjects.SubjectID = Classes.SubjectID)
          INNER JOIN Student_Schedules
              ON Student_Schedules.ClassID =
                 Classes.ClassID)
          INNER JOIN Students
              ON Students.StudentID =
              Student_Schedules.StudentID
          WHERE Student_Schedules.ClassStatus = 2 AND
              Subjects.CategoryID = 'ENG') AS S1,
         (SELECT Count(*) AS NumStudents
          FROM (Classes AS C2 INNER JOIN
               Student_Schedules AS SS3
              ON C2.ClassID = SS3.ClassID)
           INNER JOIN Subjects AS S2
              ON S2.SubjectID = C2.SubjectID
          WHERE SS3.ClassStatus = 2 And S2.CategoryID = 'ENG')
            AS StudCount
        ORDER BY S1.Grade DESC;
```

CH20_English_Student_Quintiles（18 行）

Subject ID	Stud First Name	Stud Last Name	Class Status	Grade	Category	Subject Name	RankIn Category	Num Students	Quintile
37	Scott	Bishop	2	98.07	ENG	Composition - Fundamentals	1	18	First
37	Sara	Sheskey	2	97.59	ENG	Composition - Fundamentals	2	18	First
37	John	Kennedy	2	93.01	ENG	Composition - Fundamentals	3	18	First
37	Brannon	Jones	2	91.66	ENG	Composition - Fundamentals	4	18	Second
37	Janice	Galvin	2	91.44	ENG	Composition - Fundamentals	5	18	Second
38	Kendra	Bonnicksen	2	88.91	ENG	Composition - Intermediate	6	18	Second
37	George	Chavez	2	88.54	ENG	Composition - Fundamentals	7	18	Second
37	Marianne	Wier	2	87.4	ENG	Composition - Fundamentals	8	18	Third

<< 其他行 >>

❖ **说明** 这个查询根据每门英语课的成绩将学生排名。如果一名学生修完了多门英语课，他将出现多次。要根据所有英语课的成绩将学生排名，必须先计算每位学生的平均成绩（方法是将学分与成绩的乘积总和除以总学分），再按平均成绩排名。

- **Bowling League 数据库**

“列出所有可能的球队间比赛，但不要重复同一对球队。”

要解决这个问题，需要在 FROM 子句中使用 Teams 表的两个副本。这将生成两队的所有组合，但显然你不希望出现一个球队与自己比赛的情况。为此可考虑每次处理一个球队。处理球队 1 时，你需要将其与其他所有球队 ID 更大的球队配对；处理球队 2 时，你已经将其与球队 1 配对，但将其与其他球队 ID 更大的球队配对都不会有问题。只要从球队表第二个副本中选择比第一个副本中的球队 ID 更大的球队 ID，就万事大吉！

转换/整理

Select Teams.team ID ~~from the 1st copy of the teams table~~ as team1 ID, Teams.TeamName ~~from the 1st copy of the teams table~~ as team1 name, Teams_1.team ID ~~from the 2nd copy of the teams table~~ as team1 name, and Teams_1.team name ~~from the 2nd copy of the teams table~~ as team2 name from ~~the~~ teams ~~table~~ CROSS JOIN ~~and a 2nd copy of the~~ teams ~~table~~ AS Team_1 where Teams_1.team ID ~~in the 2nd copy of the teams table is greater than~~ > Teams.team ID ~~in the 1st copy of the teams table~~ order~~ed~~ by Teams.team ID ~~in the 1st copy of the teams table~~, ~~and~~ Teams_1.team ID ~~in the 2nd copy of the teams table~~［将球队表交叉连接到球队表的第一个副本，从来自球队表第二个副本的球队 ID 大于来自球队表第一个副本的球队 ID 的行中，选择来自球队表第一个副本的球队 ID（作为球队 1ID）和球队名（作为球队 1 的名称）以及来自球队表第二个副本的球队 ID（作为球队 2ID）和球队名（作为球队 2 的名称）］

SQL

```
SELECT Teams.TeamID AS Team1ID,
  Teams.TeamName AS Team1Name,
  Teams_1.TeamID AS Team2ID,
  Teams_1.TeamName AS Team2Name
FROM Teams, Teams AS Teams_1
WHERE Teams_1.TeamID > Teams.TeamID
ORDER BY Teams.TeamID, Teams_1.TeamID;
```

CH20_Team_Pairings（45 行）

Team1ID	Team1Name	Team2ID	Team2Name
1	Marlins	2	Sharks
1	Marlins	3	Terrapins
1	Marlins	4	Barracudas
1	Marlins	5	Dolphins
1	Marlins	6	Orcas
1	Marlins	7	Manatees
1	Marlins	8	Swordfish
1	Marlins	9	Huckleberrys
1	Marlins	10	MintJuleps
2	Sharks	3	Terrapins
2	Sharks	4	Barracudas
<< 其他行 >>			

❖ **说明**　看到这个查询时你可能会问，是不是也可使用内连接并将 WHERE 子句中的查找条件移到 ON 子句中来解决这个问题呢？只要数据库系统并非只允许在 ON 子句中使用相等连接（大多数数据库系统如此），这种想法就是绝对正确的。还是那句话，对于特定的问题，使用 SQL 解决的方式总是有多种。

20.4.2　驱动表使用示例

我们接着使用驱动表来解决一些问题。下面的解决方案都使用了示例数据库中既有的驱动表。

- **Sales Orders 数据库**

“库房经理请你给库存的每件商品打印一个识别标签。”

要获悉商品的存货数量，可查看 Products 表。这里的诀窍是使用一个驱动表，这个驱动表只有一个数据类型为整

数的列，其值为 1 到存库量最大的商品的数量。示例数据库中的 ztblSeqNumbers 表就是为此而提供的。

转换/整理

Select ztblSeqNumbers.Sequence ~~from the sequence driver table~~, Products.product number ~~from the products table~~, ~~and~~ Products.product name ~~from the products table~~ from ztblSeqNumbers ~~the sequence driver table~~ CROSS JOIN ~~and the~~ products ~~table~~ where ztblSeqNumbers.sequence ~~in the sequence driver table is less than or equal to~~ <= Products.quantity on hand ~~in the products table~~ order~~ed~~ by Products.product number ~~from the products table~~, ~~and~~ ztblSequenceNumbers.sequence ~~from the sequence driver table~~（将序列号驱动表交叉连接到商品表，从序列号小于等于商品库存量的行中选择序列号、商品编号和商品名）

SQL

```
SELECT ztblSeqNumbers.Sequence,
  Products.ProductNumber,
  Products.ProductName
FROM ztblSeqNumbers, Products
WHERE ztblSeqNumbers.Sequence <=
  Products.QuantityOnHand
ORDER BY Products.ProductNumber,
  ztblSeqNumbers.Sequence;
```

CH20_Product_Stock_Labels（813 行）

Sequence	ProductNumber	ProductName
1	1	Trek 9000 Mountain Bike
2	1	Trek 9000 Mountain Bike
3	1	Trek 9000 Mountain Bike
4	1	Trek 9000 Mountain Bike
5	1	Trek 9000 Mountain Bike
6	1	Trek 9000 Mountain Bike
1	2	Eagle FS-3 Mountain Bike
2	2	Eagle FS-3 Mountain Bike
	<< 其他行 >>	

- **Entertainment Agency 数据库**

“生成一个签约日历，其中包含 2018 年 1 月的各周以及期间的演出合约。”

❖ **说明**　要找出特定时间跨度内的演出合约，需要找出开始日期不晚于该时间跨度的终点且结束日期不早于该时间跨度的起点的演出合约。要找出部分或全部在 2018 年 1 月的周，需要采取类似的措施。

转换/整理

Select ztblWeeks.week start from ~~the weeks driver table~~, ztblWeeks.week end ~~from the weeks driver table~~, Entertainers.entertainer ID ~~from the entertainers table~~, Entertainers.entertainer stage name ~~from the entertainers table~~, Customers.cust~~omer~~ first ~~name from the customers table~~, Customers.cust~~omer~~ last name ~~from the customers table~~, Engagements.start date ~~from the engagements table~~, ~~and~~ Engagements.end date ~~from the engagements table~~ from ztblWeeks ~~the weeks driver table~~ CROSS JOIN ~~and the~~ customers ~~table~~ inner join~~ed with the~~ engagements ~~table~~ on Customers.customer ID ~~in the customers table equals~~ = Engagements.customer ID ~~in the engagements table, then~~ inner join~~ed with the~~ entertainers ~~table~~ on Entertainers.entertainer ID ~~in the entertainers table equals~~ = Engagements.entertainer ID ~~in the engagements table~~ where ztblWeeks.week start ~~in the weeks driver table is less than or equal to~~ <= ‘2018-01-31’ and ztblWeeks.week end ~~in the weeks driver table is greater than or equal to~~ >= ‘2018-01-01’ and Engagements.start date ~~in the engagements table is less than or equal to~~ <= ztblWeeks.week end ~~in the weeks driver table~~ and Engagements.end date ~~in the engagements table is greater than or equal to~~ >= ztblWeeks.week start ~~in the weeks driver table~~（基于顾客 ID 相同将顾客表内连接到演出合约表，生成一个结果集，再将各周驱动表交叉连接到这个结果集；从周起始日期不晚于 2018-01-31、周结束日期不早于 2018-01-01、演出合约起始日期不晚于周结束日期、演出合约结束日期不早于周起始日期的行中，选择周起始日期、周结束日期、演唱组合 ID、演唱组合艺名、顾客名、顾客姓、演出合约起始日期、演出合约结束日期）

SQL

```
SELECT ztblWeeks.WeekStart, ztblWeeks.WeekEnd,
  Entertainers.EntertainerID,
  Entertainers.EntStageName,
  Customers.CustFirstName, Customers.
  CustLastName,
  Engagements.StartDate, Engagements.EndDate
```

```
FROM ztblWeeks, (Customers INNER JOIN Engagements
  ON Customers.CustomerID =
     Engagements.CustomerID)
INNER JOIN Entertainers
  ON Entertainers.EntertainerID =
     Engagements.EntertainerID
WHERE ztblWeeks.WeekStart <= '2018-01-31' AND
   ztblWeeks.WeekEnd >= '2018-01-01' AND
   Engagements.StartDate <= ztblWeeks.WeekEnd AND
   Engagements.EndDate >= ztblWeeks.WeekStart;
```

CH20_All_Weeks_Jan2018_All_Engagements（50 行）

Week Start	WeekEnd	Entertainer ID	EntStage Name	CustFirst Name	CustLast Name	Start Date	End Date
2017-12-31	2018-01-05	1001	Carol Peacock Trio	Mark	Rosales	2017-12-30	2018-01-08
2018-01-07	2018-01-12	1001	Carol Peacock Trio	Mark	Rosales	2017-12-30	2018-01-08
2018-01-07	2018-01-12	1001	Carol Peacock Trio	Matt	Berg	2017-01-09	2018-01-09
2018-01-21	2018-01-26	1001	Carol Peacock Trio	Dean	McCrae	2017-01-23	2018-01-31
2018-01-28	2018-02-02	1001	Carol Peacock Trio	Dean	McCrae	2017-01-23	2018-01-31
2017-12-31	2018-01-05	1002	Topazz	Estella	Pundt	2018-01-02	2018-01-10
2018-01-07	2018-01-12	1002	Topazz	Estella	Pundt	2018-01-02	2018-01-10
2017-12-31	2018-01-05	1003	JV & the Deep Six	Carol	Viescas	2017-12-31	2018-01-05
2018-01-07	2018-01-12	1003	JV & the Deep Six	Mark	Rosales	2018-01-09	2018-01-10
2018-01-28	2018-02-02	1003	JV & the Deep Six	Zachary	Ehrlich	2018-01-29	2018-02-02

<< 其他行 >>

- **School Scheduling 数据库**

“生成一个课程清单，其中包含课程所属的学期、上课日期和科目。”

❖ **说明** 这个问题有点棘手，因为在表示哪些日期上课方面，Classes 表不规范。这个表包含表示周一到周六的列，并在某天上课时，将相应的列设置为 1（真）。我需要使用驱动表 ztblSemesterDays 找出每个学期开设的课程以及课程的所有上课日期。由于有些数据库（如 Microsoft Office Access）使用–1 来表示真，而有些使用 1 来表示真，因此我将通过检查是否不等于零来确定列值是否为真。

转换/整理

Select ztblSemesterDays.semester no ~~from the semester driver table~~, ztblSemesterDays.sem~~ester~~ date ~~from the semester driver table~~, Classes.start time ~~from the classes table~~, ztblSemesterDays.sem~~ester~~ day name ~~from the semester driver table~~, Subjects.subject code ~~from the subjects table~~, Subjects.subject name ~~from the subjects table~~, Class_Rooms.building code ~~from the class rooms table~~, Class_Rooms. class room ID ~~from the class rooms table~~ from ztblSemesterDays ~~the semester driver table~~ CROSS JOIN ~~and the~~ subjects ~~table then~~ inner join~~ed with the~~ classes table on Subjects.subject ID ~~in the subjects table equals~~ = Classes.subject ID ~~in the classes table then~~ inner join~~ed with the~~ class rooms ~~table~~ on Class_Rooms.class room ID ~~in the class rooms table equals~~ = Classes.class room ID ~~in the classes table~~ where Classes.semester number ~~in the classes table equals~~ = ztblSemesterDays.semester no ~~in the semester driver table~~ and 1 ~~equals~~ = (CASE when ztblSemesterDays.semester day name ~~in the semester driver table~~ equals = 'Monday' and Classes. Monday schedule ~~in the classes table~~ does not equal <> 0 then return 1 when ztblSemesterDays.sem~~ester~~ day name ~~in the semester driver table equals~~ = 'Tuesday' and Classes.Tuesday schedule ~~in the classes table does not equal~~ <> 0 then ~~return~~ 1 when ztblSemesterDays. sem~~ester~~ day name ~~in the semester driver table equals~~ = 'Wednesday' and Classes.Wednesday schedule ~~in the classes table does not equal~~ <> 0 then return 1 when ztblSemesterDays.sem~~ester~~ day name ~~in the semester driver table equals~~ = 'Thursday' and Classes.Thursday schedule ~~in the classes table does not equal~~ <> 0 then ~~return~~ 1 when ztblSemesterDays.sem~~ester~~ day name ~~in the semester driver table equals~~ = 'Friday' and Classes.Friday schedule ~~in the classes table does not equal~~ <> 0 then ~~return~~ 1 when ztblSemesterDays. sem~~ester~~ day name ~~in the semester driver table equals~~ = 'Saturday' and Classes.Saturday schedule ~~in the classes table does not equal~~ <> 0 then ~~return~~ 1 else ~~return~~ 0 END) order~~ed~~ by ztblSemesterDays. semester no ~~in the semester driver table~~, ztblSemesterDays.sem~~ester~~ date ~~in the semester driver table~~, Subjects.subject code ~~in the subjects table~~, Class_Rooms.building code ~~in the class rooms table~~, Class_Rooms.class room ID ~~in the class rooms table~~, ~~and~~ Classes. start time ~~in the classes table~~（依次基于科目 ID 相同将科目表内连接到课程表、基于教室 ID 相同内连接到教室表，生成一个结果集，再将学期驱动表交叉连接到这个结果集。从来自学期驱动表的学期编号与来自课程表的学期编号相同且返回值为 1 的行中，选择学期编号、学期日期、开始时间、学期日期名称、科目编码、

	科目名、教学楼编码和教室 ID，并按学期编号、学期日期、科目编码、教学楼编码、教室 ID 和开始时间排序。其中返回值是这样确定的：如果来自学期驱动表的学期日期名为 Monday 且来自课程表的“周一是否上课”列不为零，就返回 1；如果来自学期驱动表的学期日期名为 Tuesday 且来自课程表的“周二是否上课”列不为零，就返回 1；如果来自学期驱动表的学期日期名为 Wednesday 且来自课程表的“周三是否上课”列不为零，就返回 1；如果来自学期驱动表的学期日期名为 Thursday 且来自课程表的“周四是否上课”列不为零，就返回 1；如果来自学期驱动表的学期日期名为 Friday 且来自课程表的“周五是否上课”列不为零，就返回 1；如果来自学期驱动表的学期日期名为 Saturday 且来自课程表的“周六是否上课”列不为零，就返回 1）

SQL

```sql
SELECT ztblSemesterDays.SemesterNo,
  ztblSemesterDays.SemDate, Classes.StartTime,
  ztblSemesterDays.SemDayName, Subjects.SubjectCode,
  Subjects.SubjectName, Class_Rooms.BuildingCode,
  Class_Rooms.ClassRoomID
FROM ztblSemesterDays, (Subjects
INNER JOIN Classes
   ON Subjects.SubjectID = Classes.SubjectID)
INNER JOIN Class_Rooms
   ON Class_Rooms.ClassRoomID =
      Classes.ClassRoomID
WHERE Classes.SemesterNumber =
      ztblSemesterDays.SemesterNo
AND Classes.StartDate <= ztblSemesterDays.SemDate
AND 1 =
(CASE WHEN ztblSemesterDays.SemDayName = 'Monday'
           AND Classes.MondaySchedule <> 0 THEN 1
      WHEN ztblSemesterDays.SemDayName = 'Tuesday'
           AND Classes.TuesdaySchedule <> 0 THEN 1
      WHEN ztblSemesterDays.SemDayName = 'Wednesday'
           AND Classes.WednesdaySchedule <> 0 THEN 1
      WHEN ztblSemesterDays.SemDayName = 'Thursday'
           AND Classes.ThursdaySchedule <> 0 THEN 1
      WHEN ztblSemesterDays.SemDayName = 'Friday'
           AND Classes.FridaySchedule <> 0 THEN 1
      WHEN ztblSemesterDays.SemDayName = 'Saturday'
           AND Classes.SaturdaySchedule <> 0 THEN 1
      ELSE 0 END)
ORDER BY ztblSemesterDays.SemesterNo,
   ztblSemesterDays.SemDate, Subjects.SubjectCode,
   Class_Rooms.BuildingCode,
   Class_Rooms.ClassRoomID,
   Classes.StartTime;
```

CH20_Class_Schedule_Calendar（7221 行）

Semester No	SemDate	Start Time	SemDay Name	Subject Code	Subject Name	Building	Class RoomID
1	2018-09-11	16:00	Monday	ACC 210	Financial Accounting Fundamentals I	IB	3305
1	2018-09-11	15:30	Monday	ART 101	Design	AS	1619
1	2018-09-11	8:00	Monday	ART 111	Drawing	AS	1627
1	2018-09-11	9:00	Monday	ART 111	Drawing	AS	1627
1	2018-09-11	11:00	Monday	ART 251	Art History	LB	1231
1	2018-09-11	14:00	Monday	ART 251	Art History	LB	1231
1	2018-09-11	10:00	Monday	BIO 100	Biological Principles	AS	1532
1	2018-09-11	12:00	Monday	BIO 101	General Biology	AS	1532
1	2018-09-11	13:30	Monday	BIO 280	Microbiology	AS	1530
1	2018-09-11	7:30	Monday	CHE 101	Chemistry	AS	1519

<< 其他行 >>

- **Bowling League 数据库**

“打印一个投球手邮寄清单，但跳过第一页开头三个已用的标签。”

> ❖ **说明** 你需要打印 3 个空的姓名和地址行，以跳过已用的标签，再打印所有投球手及其地址。你可在一条 SELECT 语句中使用 ztblSkipLabel，将所有列都替换为空格，并在该驱动表中的数字大于要跳过的标签数后停止。然后，使用 UNION ALL 与一条生成所有投球手姓名和地址的 SELECT 语句合并。你必须使用 UNION ALL，因为如果使用 UNION，将删除第一个查询生成的所有重复的空行。

转换/整理

Select ~~blanks~~ ‘ ‘ as bowler last name, ~~blanks~~ ‘ ‘ as bowler first name, ~~blanks~~ ‘ ‘ as bowler address, ~~blanks~~ ‘ ‘ as bowler city, ~~blanks~~ ‘ ‘ as bowler state, ~~blanks~~ ‘ ‘ as bowler zip from ztblSkipLabels ~~the skip labels driver table~~ where the ztblSkipLabels.label count ~~in the skip labels driver table~~ ~~is less than or equal to~~ <= 3 union~~ed with~~ all ~~rows in~~ select bowler last name, bowler first name, bowler address, bowler city, bowler state, and bowler zip from ~~the~~ bowlers ~~table~~ order~~ed~~ by bowler zip, ~~and~~ bowler last name

[在跳过标签驱动表中，从计数小于等于 3 的行中选择空格（作为投球手姓）、空格（作为投球手名）、空格（作为投球手地址）、空格（作为投球手城市）、空格（作为投球手州）、空格（作为投球手邮政编码）；从投球手表中选择投球手姓、投球手名、投球手地址、投球手城市、投球手州和投球手邮政编码，并按投球手邮政编码和投球手姓排序；使用 UNION ALL 将这两个结果集合并]

SQL

```
SELECT ' ' As BowlerLastName, ' ' As
BowlerFirstName,
  ' ' As BowlerAddress, ' ' As BowlerCity,
  ' ' As BowlerState, ' ' As BowlerZip
FROM ztblSkipLabels
WHERE ztblSkipLabels.LabelCount <= 3
UNION ALL
SELECT BowlerLastName, BowlerFirstName,
  BowlerAddress, BowlerCity, BowlerState,
   BowlerZip
FROM Bowlers
ORDER BY BowlerZip, BowlerLastName;
```

CH20_Bowler_Mailing_Skip_3（35 行）

BowlerLast Name	BowlerFirst Name	Bowler Address	Bowler City	Bowler State	Bowler Zip
Patterson	Kathryn	16 Maple Lane	Auburn	WA	98002
Patterson	Rachel	16 Maple Lane	Auburn	WA	98002
Patterson	Ann	16 Maple Lane	Auburn	WA	98002
Patterson	Neil	16 Maple Lane	Auburn	WA	98002
Patterson	Megan	16 Maple Lane	Auburn	WA	98002
Viescas	Carol	16345 NE 32nd Street	Bellevue	WA	98004
Sheskey	Sara	17950 N 59th	Seattle	WA	98011
		<< 其他行 >>			

20.5 小结

本章首先定义了非连接数据，讨论了如何在查询中使用它，并概述了交叉连接的两种用途：将主数据表彼此关联起来；将主数据表关联到驱动表。

接下来阐述了如何通过将主数据表彼此关联来解决问题，并列举了一个复杂的示例。

然后介绍了如何建立并使用驱动表，并列举了两个示例。

最后通过来自 4 个示例数据库的示例，演示了如何将主数据表彼此关联以及如何将驱动表关联到一个或多个主表。下一节列出了一些问题，建议你尽力去解决。

20.6　练习

下面列举了一些请求，以及示例数据库中对应查询的名称。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Sales Orders 数据库**
  (1) 列出各种商品每月的总销售额。
  提示：使用我提供的驱动表 ztblMonths。
  解决方案见 CH20_Product_Sales_ByMonth（253 行）。
  (2) 生成一个顾客邮寄清单，但跳过第一页开头已用的 5 个标签。
  提示：使用我提供的驱动表 ztblSeqNumbers。
  解决方案见 CH20_Customer_Mailing_Skip_5（33 行）。
  (3) 销售经理要给 2017 年 12 月消费较多的顾客发放 9 折优惠券，请使用 ztblPurchaseCoupons 表根据每位顾客在该月的消费量确定要给他多少张优惠券。
  提示：需要将 Customers 内连接到一个计算总消费量的子查询，再将前述驱动表交叉连接到生成的结果集。
  解决方案见 CH20_Customer_Dec_2017_Order_Coupons（27 行）。
  (4) 根据问题 3 的解决方案得到的结果，打印相应张数的优惠券。
  提示：使用我提供的驱动表 ztblSeqNumbers。
  解决方案见 CH20_Customer_Discount_Coupons_Print（309 行）。
  (5) 显示 2017 年和 2018 年的所有月份以及每种商品各月的销售总额。
  提示：将驱动表 ztblMonthsdriver 交叉连接到 Products 表，并使用一个子查询来获取每种商品每月的销售总额。
  解决方案见 CH20_Product_Sales_All_Months_2017_2018（960 行）。
  (6) 显示所有的商品并将它们分类——从价格实惠（Affordable）到贵得吓人（Expensive）。
  提示：交叉连接到驱动表 ztblPriceRanges。
  解决方案见 CH20_Product_Price_Ranges（40 行）。
- **Entertainment Angency 数据库**
  (1) 列出 2018 年 2 月 1 日后从未签约的经纪人和演唱组合。
  提示：交叉连接 Agents 表和 Entertainers 表，并使用一个子查询找出 2018 年 2 月 1 日后从未签约的演唱组合（在 WHERE 子句中使用 NOT IN）。
  解决方案见 CH20_Agents_Entertainers_Unbooked_Feb1_2018（162 行）。
  (2) 列出有演唱组合擅长演奏的音乐风格，并指出每种音乐风格有多少演唱组合最擅长、第二擅长、第三擅长演奏。
  提示：这类似于本章前面的查询 CH20_Customer_Style_Preference_Rankings。请使用一个子查询将擅长程度“透视”到三列中，再将 Musical_Styles 表交叉连接到这个结果集，然后计算最终结果。
  解决方案见 CH20_Entertainer_Style_Strength_Rankings（17 行）。
  (3) 确定顾客最喜欢、第二喜欢和第三喜欢的音乐风格，还有演唱组合最擅长、第二擅长和第三擅长演奏的音乐风格；将这样的顾客和演唱组合配对，即顾客最喜欢或第二喜欢的音乐风格正是演唱组合最擅长或第二擅长演奏的。
  提示：编写一个使用顾客和音乐风格表的查询，并使用 CASE 表达式透视最喜欢、第二喜欢和第三喜欢的音乐风格。你需要使用 MAX 和 GROUP BY，因为透视将把有些音乐风格的这些列的值设置为 Null。对演唱组合和音乐风格表做同样的处理，再交叉连接这两个查询，并返回顾客最喜欢或第二喜欢的音乐风格正是演唱组合最擅长或第二擅长演奏的行。

解决方案见 CH20_Customers_Match_ Entertainers_FirstSecond_PrefStrength（6 行）。

(4) 列出所有的月份并计算每个演唱组合各月的收入。

提示：使用驱动表 ztblMonths 透视每月的收入，再使用 SUM 计算演唱组合每月的总收入。

解决方案见 CH20_Entertainer_BookingAmount_ByMonth（12 行）。

(5) 显示 2017 年 12 月的每一天以及在该天有演出的演唱组合。

提示：编写一个子查询，将演唱组合表内连接到演出合约表，生成一个结果集。将驱动表 ztblDays 交叉连接到这个结果集，再左外连接到 ztblDays，以得到所有的日期。

解决方案见 CH20_All_December_Days_Any_Bookings（79 行）。

(6) 生成一个顾客邮寄清单，但跳过第一页开头已用的 4 个标签。

解决方案见 CH20_Customer_Mailing_Skip_4（19 行）。

- **School Scheduling 数据库**

(1) 列出所有的学生及其可选的课程，将已注册或修完的课程排除在外。务必同时列出先修课。

提示：将科目表内连接到课程表，生成一个结果集，再将学生表交叉连接到这个结果集。使用子查询在学生选课表中找出学生 ID 与当前学生的 ID 相同且课程状态为 1（已注册）或 2（已结束）的课程，并将它们排除在外。

解决方案见 CH20_Students_Additional_Courses（1894 行）。

(2) 按性别和婚姻状况分州的学生数量。

提示：使用驱动表 ztblGenderMatrix 和 ztblMaritalStatusMatrix 进行透视。

解决方案见 CH20_Student_Crosstab_Gender_MaritalStatus（4 行）。

(3) 使用驱动表 ztblProfRatings 中的数据计算每位教员讲授的课程的平均评分，还有全体教员的平均评分。

解决方案见 CH20_Staff_Proficiency_Ratings（24 行）。

(4) 创建一个学生邮寄清单，跳过第一页开头两个已用的标签。

解决方案见 CH20_Student_Mailing_Skip_2（20 行）。

- **Bowling League 数据库**

(1) 显示每个投球手，以及使用驱动表 ztblBowlerRatings 根据平均原始得分确定的等级。

解决方案见 CH20_Bowler_Ratings（32 行）。

(2) 列出 2017 年 9~12 月的每一周，以及安排在该周举行的联赛的比赛地点。

解决方案见 CH20_Tournament_Week_ Schedule_2017（19 行）。

第 21 章 执行复杂的分组计算

“一个团体要进入开放社会，必须先团结一致。”

——斯托克利·卡迈克

本章涵盖如下主题：

- ❑ 不同分组的合计
- ❑ 扩展 GROUP BY 子句
- ❑ 使用 ROLLUP 获取分层合计
- ❑ 使用 CUBE 计算各种组合的汇总
- ❑ 使用 GROUPING SETS 合并汇总
- ❑ 分组技术变种
- ❑ 语句举例
- ❑ 小结
- ❑ 练习

第四部分介绍了如何对数据进行汇总和分组。具体地说，第 12 章介绍了如何计算总计、计数和其他聚合值；第 13 章介绍了如何将数据分组；第 14 章介绍了如何筛选分组数据以及如何筛选出要分组的数据。然而，我发现在报告数据方面，这存在一定的局限性：每次只能以一种方式分组，在有些情况下这根本不够。

本章介绍如何在分组方面更进一步。

21.1 不同分组的合计

第 13 章介绍了如何使用 GROUP BY 按一列或多列对数据进行分组。按多列分组时，将根据这些列的每种值组合执行聚合计算。

我们来看看学生分布情况。使用第 13 章介绍的方法，可编写查询，根据州、性别和婚姻状况来汇总数据，相应的 SQL 如下：

```
SQL         SELECT StudState, StudGender, StudMaritalStatus,
              Count(*) AS Number
            FROM Students
            GROUP BY StudState, StudGender, StudMaritalStatus;
```

结果类似于下表（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_GROUP_BY）。

StudState	StudGender	StudMaritalStatus	Number
CA	F	W	1
CA	M	S	1
OR	F	M	1
OR	F	S	1
OR	M	S	2
TX	F	S	2
TX	M	S	1
WA	F	D	1
WA	F	M	1
WA	F	S	3
WA	M	S	4

这是一种很有用的数据查看方式，但如果我还想知道各州的学生总数或女学生总数，该怎么办呢？为了获取其他细分情况，必须再编写一个或多个查询。

如果有另一种数据检索方式，让我能够返回类似于下面的结果就好了。

State	Gender	MaritalStatus	Number
Any State	Any Gender	Any Status	18
Any State	Any Gender	D	1
Any State	Any Gender	M	2
Any State	Any Gender	S	14
Any State	Any Gender	W	1
Any State	F	Any Status	10
Any State	F	D	1
Any State	F	M	2
Any State	F	S	6
Any State	F	W	1
Any State	M	Any Status	8
Any State	M	S	8
CA	Any Gender	Any Status	2
CA	Any Gender	S	1
CA	Any Gender	W	1
CA	F	Any Status	1
CA	F	W	1
CA	M	Any Status	1
CA	M	S	1
OR	Any Gender	Any Status	4
OR	Any Gender	M	1
OR	Any Gender	S	3
OR	F	Any Status	2
OR	F	M	1
OR	F	S	1
OR	M	Any Status	2
OR	M	S	2
TX	Any Gender	Any Status	3

（续）

State	Gender	MaritalStatus	Number
TX	Any Gender	S	3
TX	F	Any Status	2
TX	F	S	2
TX	M	Any Status	1
TX	M	S	1
WA	Any Gender	Any Status	9
WA	Any Gender	D	1
WA	Any Gender	M	1
WA	Any Gender	S	7
WA	F	Any Status	5
WA	F	D	1
WA	F	M	1
WA	F	S	3
WA	M	Any Status	4
WA	M	S	4

针对这个表可能需要做些说明。请看其中的灰色行，它们是前面的查询返回的 11 行，指出了州、性别和婚姻状况的每种组合对应的学生数量。我们来看看其他行，这些行有一列或多列为所有可能取值。例如，第一行是学生的总数（18），不管他们来自哪个州，以及性别和婚姻状况如何。接下来的一行是离异学生总数（1），不管他们来自哪个州、性别如何。第 6 行是女学生总数（10），不管她们来自哪个州、婚姻状况如何。

在进行学生人口统计时，这是不是很有用？本章后面将介绍如何获得这样的结果（如果你好奇，想知道相应的 SQL，请查看示例数据库 Student Scheduling 中的查询 CH21_Students_State_Gender_MaritalStatus_Count_CUBE_No_Nulls）。

21.2　扩展 GROUP BY 子句

本章会介绍更全面的分组，但第 13 章介绍的知识也适用。事实上，如果你查看更全面的分组的语法，将发现它们几乎与第 13 章的语法图相同。

语法

我们来仔细研究一下完整的 GROUP BY 子句。图 21-1 显示了 SELECT 语句的基本语法图，但对 GROUP BY 进行了扩展，以显示本章将介绍的功能。如果你将该图与图 13-1 进行比较，将发现差别在于这里新增了三个关键字（ROLLUP、CUBE 和 GROUPING SETS），且使用了这些关键字时，列引用放在括号内。

你可能猜到了，这些关键字将导致返回不同的结果。

❖ **说明**　GROUP BY 扩展不过是 GROUP BY 的变种，因此第 13 章谈论的所有限制在这里也适用。具体地说，对于在 SELECT 子句中指定的列，如果不是聚合表达式的一部分，它就必须包含在 GROUP BY 子句中；GROUP BY 子句只能引用 FROM 和 WHERE 生成的列，而不能引用 SELECT 子句中创建的表达式。

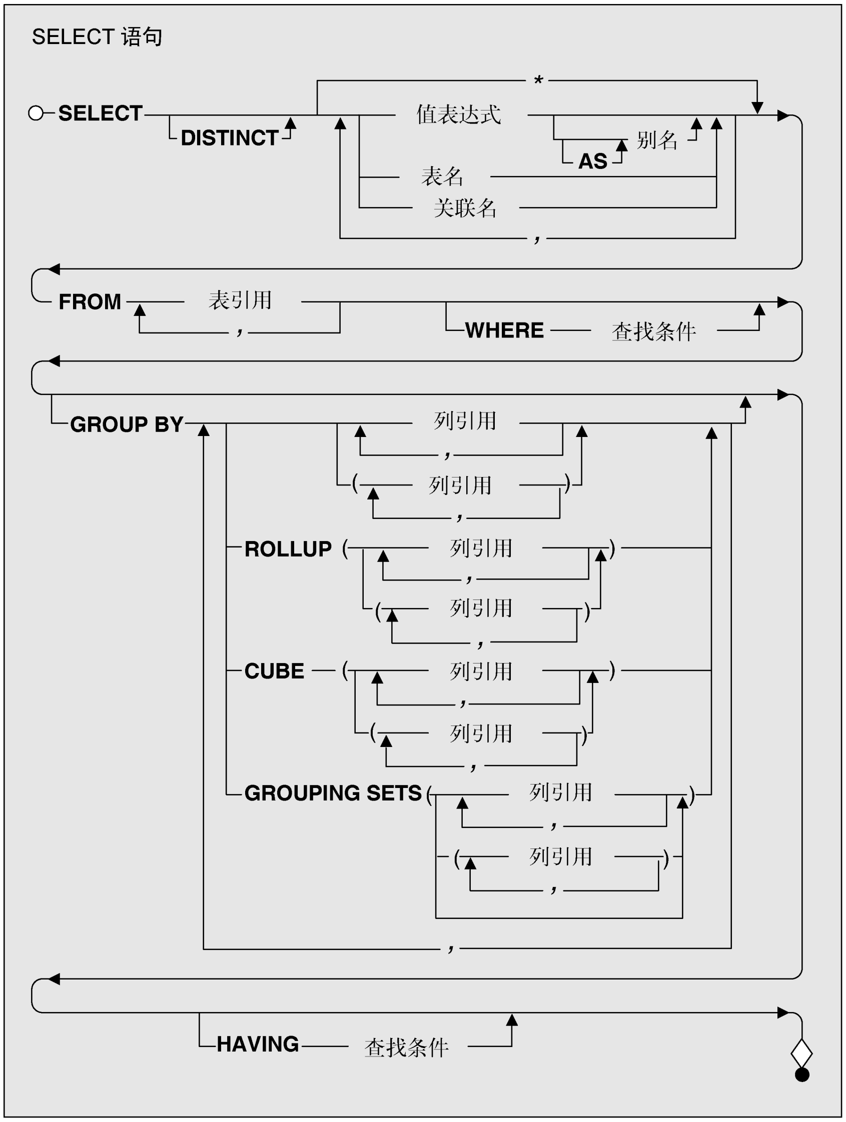


图 21-1　GROUP BY 子句的完整语法

21.3　使用 ROLLUP 获取分层合计

你知道，使用 GROUP BY 时，将针对指定列的每个不同的值组合汇总数据。但添加 ROLLUP 后，数据库系统不仅会生成分组小计，还会生成分层合计及总计。如果在 ROLLUP 中指定了 n 列，就有 $n+1$ 层汇总数据。请注意，合计是按从右到左的顺序进行的，因此在 ROLLUP 中指定列的顺序很重要。

❖ **说明**　所有示例语句和解决方案都可在相应的示例数据库——Sales Orders、Entertainment Agency、School Scheduling 和 Bowling League 中找到。由于 Microsoft Access 和 MySQL 都不支持 GROUPING SETS，因此只有 Microsoft SQL Server 和 PostgreSQL 示例数据库中有解决方案。

假设要按州、性别和婚姻状况对学生计算小计，并按州以及州和性别计算合计，请求可能类似于下面这样。

> ❖ **说明**　贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”流程。考虑到你现在对这个流程已非常熟悉，为简化流程，下面的示例中都将转换和整理合二为一了。

“显示州、性别和婚姻状况的每种不同的组合对应的学生数，州和性别的每种不同的组合对应的学生数，以及每个州的学生总数。”

转换/整理　Select ~~the~~ stud~~ent~~ state, stud~~ent~~ gender, stud~~ent~~ marital status, ~~and the~~ count (*) ~~of rows~~ from ~~the~~ Students ~~table,~~ group~~ed~~ by ~~and~~ roll~~ed~~ up ~~by~~ stud~~ent~~ state, stud~~ent~~ gender, ~~and~~ stud~~ent~~ marital status（在学生表中，按州、性别和婚姻状况分组并累积，并选择州、性别、婚姻状况和行数）

SQL
```
SELECT StudState, StudGender, StudMaritalStatus,
  Count(*) AS Number
FROM Students
GROUP BY ROLLUP
  (StudState, StudGender, StudMaritalStatus);
```

> ❖ **说明**　需要计算用于分组的部分而不是全部列的不同组合的合计时，应使用关键字 ROLLUP。

这个查询的结果类似于下表(这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_ROLLUP_Order1)。

StudState	StudGender	StudMaritalStatus	Number
CA	F	W	1
CA	F	NULL	1
CA	M	S	1
CA	M	NULL	1
CA	NULL	NULL	2
OR	F	M	1
OR	F	S	1
OR	F	NULL	2
OR	M	S	2
OR	M	NULL	2
OR	NULL	NULL	4
TX	F	S	2
TX	F	NULL	2
TX	M	S	1
TX	M	NULL	1
TX	NULL	NULL	3
WA	F	D	1
WA	F	M	1
WA	F	S	3
WA	F	NULL	5
WA	M	S	4
WA	M	NULL	4
WA	NULL	NULL	9
NULL	NULL	NULL	18

我们来看看结果。在特定行中，如果一列为 Null，可将其解读为涵盖了所有的值。

> ❖ **说明**　本书前言提醒过，在你使用的数据库系统中，行的排列顺序不一定与本书示例中显示的相同——除非使用 ORDER BY 子句。即便指定了排序方式，在数据库系统返回的结果中，对于 ORDER BY 子句中未包含的列，根据它们的排列顺序也可能不同，这是因为优化方法因数据库系统而异。
>
> 如果你在 Microsoft SQL Server 中运行示例，则从视图中选择行时，将不会遵循其中的 ORDER BY 子句指定的排列顺序。要遵循 ORDER BY 子句指定的排列顺序，必须打开视图的设计，并在显示界面中执行它。
>
> 另外，当你使用 GROUP BY 子句时，数据库系统返回的结果集看起来像是根据该子句中指定的列进行了排序。这是因为有些优化器首先会在内部对数据排序，以提高 GROUP BY 的处理速度。别忘了，如果要按特定的顺序排列，必须使用 ORDER BY 子句。

第一行表明，有一名丧夫的女学生，她来自加州。由于没有其他来自加州的女学生，因此第二行汇总，指出有一名女学生来自加州（别忘了，StudMaritalStatus 列的 Null 表示“任何婚姻状况”）。第三行表明，有一名来自加州的单身男学生；同样，由于没有其他来自加州的男学生，因此第四行汇总，指出有一名来自加州的男学生。涵盖所有性别和婚姻状况后，第五行汇总，指出有两名学生来自加州。

第 6 行和第 7 行指出来自俄勒冈州的已婚和单身女学生各有一名。这是来自俄勒冈州的全部女学生，因此第 8 行汇总，指出有两名女学生来自俄勒冈州。

这种汇总不断继续，与分组结果交替出现，直到最后一行，它执行汇总，指出总共有 18 名学生。

由于在 ROLLUP 子句中指定了 3 列（依次为 StudState、StudGender 和 StudMaritalStatus），因此有 4 层小计和合计：

- StudState、StudGender 和 StudMaritalStatus 的不同组合；
- StudState 和 StudGender 的不同组合（不考虑 StudMaritalStatus 的值）；
- StudState 的不同值，不考虑 StudGender 和 StudMaritalStatus 的值；
- 总计。

> ❖ **说明**　对于当前累积的列，这个查询将其值显示为 Null。如果你愿意，可使用 CASE 表达式、ISNULL 函数或 COALESCE 函数（SQL Server），或者 CASE 表达式或 COALESCE 函数（PostgreSQL），将 Null 值转换为更有意义的内容。

用户可能不想看到 Null。对于当前累积的列，这个查询将其值显示为 Null，但如果表中包含 NULL 值，就不会这样（这里没有这样的列）。因此，可使用 CASE 表达式、ISNULL 函数或 COALESCE 函数（SQL Server），或者 CASE 表达式或 COALESCE 函数（PostgreSQL），将 Null 值转换为更有意义的内容。但有一种更好的办法，那就是使用函数 GROUPING，如图 21-2 所示。

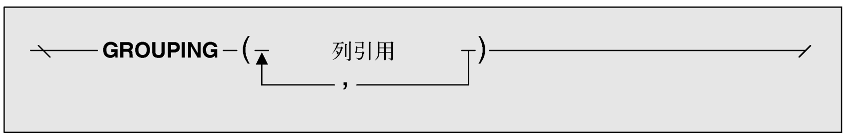


图 21-2　函数 GROUPING

这是一个很方便的小函数，用于被 CASE、ROLLUP 或 GROUPING SETS 分组的列时，它返回一个表示分组层级的数字。如果返回的值为零，就表明没有累积，因此可显示当前列的值；如果返回的值不是零，就说明当前列被累积，肯定包含的是 Null 值。你可在 CASE 表达式（参见第 19 章）中使用这个函数来确定该显示列值还是字面量值 Any。

因此，可换种说法，这样陈述前面的问题：

> “显示州、性别和婚姻状况的每种不同的组合对应的学生数，州和性别的每种不同的组合对应的学生数，以及每个州的学生总数。在累积行中，显示 Any State、Any Gender 或 Any Status。”

转换/整理

Select ~~when~~ (CASE WHEN ~~the grouping level of~~ GROUPING(student state) ~~is~~ = 0 then ~~display~~ student state else ‘Any State’ END), ~~when~~ (CASE WHEN ~~the grouping level of~~ GROUPING(student gender) ~~is~~ = 0 then ~~display~~ student gender else ‘Any Gender’ END), ~~when~~ (CASE WHEN ~~the grouping level of~~ GROUPING(student marital status) ~~is~~ = 0 then ~~display~~ student marital status else ‘Any Status’ END), ~~and the~~ count(*) ~~of rows~~ from ~~the~~ Students ~~table~~, grouped by ~~and~~ roll~~ed~~ up ~~by~~ student state, student gender, ~~and~~ student marital status（在学生表中，按州、性别和婚姻状况分组并累积，并选择州、性别、婚姻状况和行数；对于州、性别和婚姻状况，使用 GROUPING 检查其分组层级，如果不为 0，就分别显示 Any State、Any Gender 和 Any Status）

SQL

```
SELECT (CASE WHEN GROUPING(StudState) = 0
             THEN StudState
             ELSE 'Any State' END) AS State,
    (CASE WHEN GROUPING(StudGender) = 0
          THEN StudGender
          ELSE 'Any Gender') AS Gender,
    (CASE WHEN GROUPING(StudMaritalStatus) = 0
          THEN StudMaritalStatus
          ELSE 'Any Status') AS MaritalStatus,
    Count(*) AS Number
FROM Students
GROUP BY ROLLUP (StudState, StudGender,
  StudMaritalStatus);
```

这个查询返回的结果集如下表所示（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_ROLLUP_No_Nulls）。

State	Gender	MaritalStatus	Number
CA	F	W	1
CA	F	Any Status	1
CA	M	S	1
CA	M	Any Status	1
CA	Any Gender	Any Status	2
OR	F	M	1
OR	F	S	1
OR	F	Any Status	2
OR	M	S	2
OR	M	Any Status	2
OR	Any Gender	Any Status	4
TX	F	S	2
TX	F	Any Status	2
TX	M	S	1
TX	M	Any Status	1
TX	Any Gender	Any Status	3
WA	F	D	1
WA	F	M	1
WA	F	S	3
WA	F	Any Status	5
WA	M	S	4
WA	M	Any Status	4
WA	Any Gender	Any Status	9
Any State	Any Gender	Any Status	18

这个查询返回 4 级小计和合计：

- ❑ StudState、StudGender 和 StudMaritalStatus 的不同组合；
- ❑ StudState 和 StudGender 的不同组合（不考虑 StudMaritalStatus 的值）；
- ❑ StudState 的不同值，不考虑 StudGender 和 StudMaritalStatus 的值；
- ❑ 总计。

❖ **说明** 这个查询演示了这样一个规则，即 GROUP BY 子句必须引用 FROM 和 WHERE 生成的列，而不能引用 SELECT 子句中的表达式。如你所见，在 SELECT 子句中，给列指定了别名，但在 GROUP BY 子句中，不能使用这些别名，而必须使用原始列名。

下面来看看在 ROLLUP 子句中以不同的顺序指定列时的结果。请看下面的请求：

“显示婚姻状况、性别和州的每种不同的组合对应的学生数，婚姻状况和性别的每种不同的组合对应的学生数，以及每种婚姻状况的学生总数。”

转换/整理	Select ~~the~~ stud~~ent~~ marital status, stud~~ent~~ gender, stud~~ent~~ state, ~~and the~~ count(*) ~~of rows~~ from ~~the~~ Students ~~table~~, group~~ed~~ by ~~and~~ roll~~ed~~ up ~~by~~ student marital status, student gender, and student state（在学生表中，按婚姻状况、性别和州分组并累积，并选择婚姻状况、性别、州和行数）

SQL

```
SELECT StudMaritalStatus, StudGender, StudState,
  Count(*) AS Number
FROM Students
GROUP BY ROLLUP
  (StudMaritalStatus, StudGender, StudState);
```

返回的结果类似于下表（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_ROLLUP_Order2）。

StudMaritalStatus	StudGender	StudState	Number
D	F	WA	1
D	F	NULL	1
D	NULL	NULL	1
M	F	OR	1
M	F	WA	1
M	F	NULL	2
M	NULL	NULL	2
S	F	OR	1
S	F	TX	2
S	F	WA	3
S	F	NULL	6
S	M	CA	1
S	M	OR	2
S	M	TX	1
S	M	WA	4
S	M	NULL	8
S	NULL	NULL	14
W	F	CA	1
W	F	NULL	1
W	NULL	NULL	1
NULL	NULL	NULL	18

这个查询不像前面的查询那样返回 24 行，而是返回 21 行。从前面的查询结果很容易看出，在总共 18 名学生中，有 2 名来自加州、4 名来自俄勒冈州、3 名来自得克萨斯州、9 名来自华盛顿州；而现在很容易看出有 1 名学生离异、2 名已婚、14 名单身、1 名丧偶。

由于在 ROLLUP 中，列出这 3 列的顺序依次为 StudMaritalStatus、StudGender 和 StudState，因此 4 层小计和合计如下：

❑ StudMaritalStatus、StudGender 和 StudState 的不同组合；

- ❑ StudMaritalStatus 和 StudGender 的不同组合（不管 StudState 的值是什么）；
- ❑ StudMaritalStatus 的不同值（不管 StudGender 和 StudState 的值是什么）；
- ❑ 总计。

请注意，现在的焦点是婚姻状况，因此在 ROLLUP 中将它放在最前面。放在第二位的是性别，因此返回了婚姻状况和性别的每种不同的组合对应的合计；最后添加的州生成婚姻状况、性别和州的每种不同的组合对应的小计。汇总时，依次累积州、性别和婚姻状况，最终累积的结果是总行数。

如果请求是这样的：

“显示州、性别和婚姻状况的每种不同的组合对应的学生数，婚姻状况和性别的每种不同的组合对应的学生数，以及每种婚姻状况的学生数。”

转换/整理　Select ~~the~~ student state, stud~~ent~~ gender, stud~~ent~~ marital status, ~~and the~~ count(*) ~~of rows~~ from ~~the~~ Students ~~table,~~ group~~ed~~ by ~~and~~ roll~~ed~~ up ~~by~~ student marital status, student gender, and student state（在学生表中，按婚姻状况、性别和州分组并累积，并选择州、性别、婚姻状况和行数）

SQL
```
SELECT StudState, StudGender, StudMaritalStatus,
  Count(*) AS Number
FROM Students
GROUP BY ROLLUP
  (StudMaritalStatus, StudGender, StudState);
```

我只是调整了 SELECT 子句中列的排列顺序。返回的结果类似于下表（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_ROLLUP_Order3）。

StudState	StudGender	StudMaritalStatus	Number
WA	F	D	1
NULL	F	D	1
NULL	NULL	D	1
OR	F	M	1
WA	F	M	1
NULL	F	M	2
NULL	NULL	M	2
OR	F	S	1
TX	F	S	2
WA	F	S	3
NULL	F	S	6
CA	M	S	1
OR	M	S	2
TX	M	S	1
WA	M	S	4
NULL	M	S	8
NULL	NULL	S	14
CA	F	W	1
NULL	F	W	1
NULL	NULL	W	1
NULL	NULL	NULL	18

依然返回了 21 行，小计和合计依然是下面 4 层：

- ❑ StudMaritalStatus、StudGender 和 StudState 的不同组合；
- ❑ StudMaritalStatus 和 StudGender 的不同组合（不管 StudState 的值是什么）；
- ❑ StudMaritalStatus 的不同值（不管 StudGender 和 StudState 的值是什么）；
- ❑ 总计。

然而，由于我在 SELECT 子句中首先列出的是 StudState，因此在输出中它在最前面。

❖ **说明**　需要指出的是，MySQL 其实支持 ROLLUP 扩展（MySQL 确实不支持 CUBE 和 GROUPING SETS 扩展），但语法不同，这就是我没有提供 MySQL 示例查询的原因。有关这方面的详细信息，请参阅 MySQL 文档。

21.4　使用 CUBE 计算各种组合的汇总

你知道，ROLLUP 按从右到左的顺序生成合计和总计。CUBE 扩展除了生成这些汇总值外，还会生成指定列不同组合的小计。如果在 CUBE 中指定了 *n* 列，将生成 2^n 种汇总。

假设有这样一个请求：

“显示州、性别和婚姻状况的各种组合对应的学生数，所有这些组合对应的学生总数，州和性别的不同组合、州和婚姻状况的不同组合、性别和婚姻状况的不同组合对应的学生数，以及每个州、每种性别和每种婚姻状况对应的学生数。”

转换/整理　Select ~~the~~ stud~~ent~~ state, stud~~ent~~ gender, stud~~ent~~ marital status, ~~and the~~ count(*) ~~of rows~~ from ~~the~~ Students ~~table~~, ~~summarized in sets~~ GROUP BY CUBE student state, student gender, ~~and~~ student marital status ~~and by all combinations of pairs of student state, student gender and student marital status~~（在学生表中，使用 GROUP BY CUBE 按州、性别、婚姻状况中的零、一、二和三个分组，从各个分组中选择州、性别、婚姻状况和行数）

SQL

```
SELECT StudState, StudGender, StudMaritalStatus,
  Count(*) AS Number
FROM Students
GROUP BY CUBE
  (StudState, StudGender, StudMaritalStatus);
```

❖ **说明**　需要计算每种数目的列的各种组合对应的汇总时，应使用关键字 CUBE。

结果类似于下表（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_CUBE_Order1）。

StudState	StudGender	StudMaritalStatus	Number
WA	F	D	1
NULL	F	D	1
NULL	NULL	D	1
OR	F	M	1
WA	F	M	1
NULL	F	M	2
NULL	NULL	M	2
OR	F	S	1
TX	F	S	2
WA	F	S	3
NULL	F	S	6
CA	M	S	1
OR	M	S	2
TX	M	S	1
WA	M	S	4
NULL	M	S	8
NULL	NULL	S	14
CA	F	W	1

（续）

StudState	StudGender	StudMaritalStatus	Number
NULL	F	W	1
NULL	NULL	W	1
NULL	NULL	NULL	18
CA	NULL	S	1
CA	NULL	W	1
CA	NULL	NULL	2
OR	NULL	M	1
OR	NULL	S	3
OR	NULL	NULL	4
TX	NULL	S	3
TX	NULL	NULL	3
WA	NULL	D	1
WA	NULL	M	1
WA	NULL	S	7
WA	NULL	NULL	9
CA	F	NULL	1
OR	F	NULL	2
TX	F	NULL	2
WA	F	NULL	5
NULL	F	NULL	10
CA	M	NULL	1
OR	M	NULL	2
TX	M	NULL	1
WA	M	NULL	4
NULL	M	NULL	8

在 CUBE 中指定了 3 列，因此数据库系统将生成 2^3（8）种组合对应的汇总：

- ❑ StudState、StudGender 和 StudMaritalStatus 的不同组合；
- ❑ StudState 和 StudGender 的不同组合（不管 StudMaritalStatus 的值是什么）；
- ❑ StudState 和 StudMaritalStatus 的不同组合（不管 StudGender 的值是什么；
- ❑ StudGender 和 StudMaritalStatus 的不同组合（不管 StudState 的值是什么）；
- ❑ StudState 的不同值（不管 StudGender 和 StudMaritalStatus 的值是什么）；
- ❑ StudGender 的不同值（不管 StudState 和 StudMaritalStatus 的值是什么）；
- ❑ StudMaritalStatus 的不同值（不管 StudState 和 StudGender 的值是什么）；
- ❑ 总计。

CUBE 生成指定列的各种组合对应的汇总，因此你可能猜到了，在 SQL 语句中调整列的顺序对结果没有影响。如果前面的请求是这样的：

"显示婚姻状况、性别和州的各种组合对应的学生数，所有这些组合对应的学生总数，州和性别的不同组合、州和婚姻状况的不同组合、性别和婚姻状况的不同组合对应的学生数，以及每个州、每种性别和每种婚姻状况的学生数。"

转换/整理　Select ~~the~~ stud~~ent~~ marital status, stud~~ent~~ gender, stud~~ent~~ state, ~~and the~~ count(*) ~~of rows~~ from ~~the~~ Students ~~table, summarized in sets~~ GROUP BY CUBE stud~~ent~~ marital status, stud~~ent~~ gender, ~~and~~ student state ~~and by all combinations of pairs of student marital status, student gender and student state~~（在学生表中，使用 GROUP BY CUBE 按婚姻状况、性别、州中的零、一、二和三个分组，从各个分组中选择州、性别、婚姻状况和行数）

```
SQL         SELECT StudMaritalStatus, StudGender, StudState,
              Count(*) AS Number
            FROM Students
            GROUP BY CUBE
              (StudMaritalStatus, StudGender, StudState);
```

结果如下表所示（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_CUBE_Order2）。

StudMaritalStatus	StudGender	StudState	Number
W	F	CA	1
NULL	F	CA	1
S	M	CA	1
NULL	M	CA	1
NULL	NULL	CA	2
M	F	OR	1
S	F	OR	1
NULL	F	OR	2
S	M	OR	2
NULL	M	OR	2
NULL	NULL	OR	4
S	F	TX	2
NULL	F	TX	2
S	M	TX	1
NULL	M	TX	1
NULL	NULL	TX	3
D	F	WA	1
M	F	WA	1
S	F	WA	3
NULL	F	WA	5
S	M	WA	4
NULL	M	WA	4
NULL	NULL	WA	9
NULL	NULL	NULL	18
D	NULL	WA	1
D	NULL	NULL	1
M	NULL	OR	1
M	NULL	WA	1
M	NULL	NULL	2
S	NULL	CA	1
S	NULL	OR	3
S	NULL	TX	3
S	NULL	WA	7
S	NULL	NULL	14
W	NULL	CA	1
W	NULL	NULL	1
D	F	NULL	1
M	F	NULL	2
S	F	NULL	6
W	F	NULL	1
NULL	F	NULL	10
S	M	NULL	8
NULL	M	NULL	8

如果你仔细查看这两个结果集，将发现它们包含的 43 行是相同的，唯一不同的是列和行的排列顺序。

21.5　使用 GROUPING SETS 合并汇总

第三种 GROUP BY 扩展是 GROUPING SETS。可以想见，使用 CUBE 计算各种组合对应的汇总可能需要消耗大量资源（在数据包含多维时尤其如此），但你可能并非对所有的汇总都感兴趣。如果不需要所有的汇总，但 GROUP BY 或 GROUP BY ROLLUP 又不能满足需求，可能就该使用 GROUPING SETS 了。

GROUPING SETS 的效果类似于使用 UNION 合并多个不同 GROUP BY 查询的结果。

在最简单的情况下，请求可能是这样的：

“显示每个州、每种性别和每种婚姻状况的学生数。”

转换/整理　Select ~~the~~ student state, student gender, student marital status, ~~and the~~ count(*) ~~of rows~~ from ~~the~~ Students ~~table~~, group~~ed~~ by ~~and in~~ GROUPING SETS ~~of~~ student state, student gender, ~~and~~ student marital status（在学生表中，使用 GROUPING SETS 分别按州、性别和婚姻状况分组，从各个分组中选择州、性别、婚姻状况和行数）

SQL
```
SELECT StudState, StudGender, StudMaritalStatus,
  Count(*) AS Number
FROM Students
GROUP BY GROUPING SETS
  (StudState, StudGender, StudMaritalStatus);
```

这将返回如下表所示的结果（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_GROUPING_SETS）。

StudState	StudGender	StudMaritalStatus	Number
NULL	NULL	D	1
NULL	NULL	M	2
NULL	NULL	S	14
NULL	NULL	W	1
NULL	F	NULL	10
NULL	M	NULL	8
CA	NULL	NULL	2
OR	NULL	NULL	4
TX	NULL	NULL	3
WA	NULL	NULL	9

如果你使用的数据库不支持 GROUPING SETS，可使用运算符 UNION 合并三个不同的 GROUP BY 查询来获得相同的结果，如下所示。

SQL
```
SELECT NULL AS StudState, NULL AS StudGender,
   StudMaritalStatus, Count(*) AS Number
FROM Students
GROUP BY StudMaritalStatus
UNION
SELECT NULL, StudGender, NULL, Count(*)
FROM Students
GROUP BY StudGender
UNION
SELECT StudState, NULL, NULL, Count(*)
FROM Students
GROUP BY StudState;
```

（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_GROUP_BY_UNION）。

GROUPING SETS 的真正威力在于，它提供了灵活性，让你能够指定要获取哪些汇总。

请看下面的请求：

“显示不同州的学生数、州和性别的不同组合对应的学生数，以及州和婚姻状况的不同组合对应的学生数，但不显示学生总数。”

> ❖ **说明**　需要计算一个和多个列的不同组合对应的汇总，但不需要计算所有列的不同组合对应的汇总时，应使用 GROUPING SETS。在不需要计算总计时尤其如此，因为 CUBE 和 ROLLUP 都返回总计，而 GROUPING SETS 不会——除非你在 GROUPING SETS 列表中指定了一个空集。

转换/整理　Select ~~the~~ student state, student gender, student marital status, ~~and the~~ count(*) ~~of rows~~ from ~~the~~ Students ~~table,~~ group~~ed~~ by ~~and in~~ GROUPING SETS ~~of the combination of~~ student state ~~and~~ student gender, ~~and by the combination of~~ student state ~~and~~ student marital status（在学生表中，分别按州、州与性别以及州与婚姻状况分组，从各个分组中选择州、性别、婚姻状况和行数）

SQL

```
SELECT StudState, StudGender, StudMaritalStatus,
  Count(*) AS Number
FROM Students
GROUP BY GROUPING SETS
  (StudState,
   (StudState, StudGender),
   (StudState, StudMaritalStatus));
```

这将返回如下表所示的结果（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_GROUPING_SETS_1）。

StudState	StudGender	StudMaritalStatus	Number
CA	NULL	S	1
CA	NULL	W	1
CA	NULL	NULL	2
OR	NULL	M	1
OR	NULL	S	3
OR	NULL	NULL	4
TX	NULL	S	3
TX	NULL	NULL	3
WA	NULL	D	1
WA	NULL	M	1
WA	NULL	S	7
WA	NULL	NULL	9
CA	F	NULL	1
OR	F	NULL	2
TX	F	NULL	2
WA	F	NULL	5
CA	M	NULL	1
OR	M	NULL	2
TX	M	NULL	1
WA	M	NULL	4

如果你仔细查看这些结果，将发现三种不同的组合对应的汇总：

- ❑ StudState 的不同值（不管 StudGender 和 StudMaritalStatus 的值是什么）；
- ❑ StudState 和 StudGender 的不同组合（不管 StudMaritalStatus 的值是什么）；
- ❑ StudState 和 StudMaritalStatus 的不同组合（不管 StudGender 的值是什么）。

请注意，ROLLUP 按从右到左的顺序累积，CUBE 计算各种组合对应的汇总，而 GROUPING SETS 不同，它让你能够准确地指定要计算哪些组合对应的汇总。

通过在 GROUP BY 后面指定一列，再在 CUBE 中指定余下的列，可获得几乎相同的结果，这将在下一节介绍。

21.6　分组技术变种

只要仔细查看图 21-1 就知道，在 GROUP BY 子句中，完全可以将 ROLLUP、CUBE 或 GROUPING SETS 与简单分组结合起来使用。另外，在 ROLLUP、CUBE 和 GROUPING SETS 中，指定的列列表可包含列组合。坦率地说，我想不出需要结合使用 ROLLUP、CUBE 和 GROUPING SETS 的情形，但将一列或多列“提升”为简单分组列，以及在分组集中指定列子集是合理的。我们来看两个示例。

假设你不需要所有的汇总，请求可能类似于下面这样：

“分别按州以及性别和婚姻状况分组，并计算各个分组的小计。”

转换/整理　Select ~~the~~ student state, student gender, student marital status, ~~and the~~ count(*) ~~of rows~~ from ~~the~~ Students ~~table,~~ group~~ed~~ by ~~and in a~~ GROUPING SETS ~~set of~~ student state, student gender, ~~and~~ student marital status（在学生表中，使用 GROUPING SETS 分别按州以及性别和婚姻状况分组，从每个小组中选择州、性别、婚姻状态和行数）

SQL
```
SELECT StudState, StudGender, StudMaritalStatus,
  Count(*) AS Number
FROM Students
GROUP BY GROUPING SETS (StudState,
  (StudGender, StudMaritalStatus));
```

返回的结果类似于下表（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_ Students_State_Gender_MaritalStatus_Count_GROUPING_SETS_2）。

StudState	StudGender	StudMaritalStatus	Number
NULL	F	D	1
NULL	F	M	2
NULL	F	S	6
NULL	M	S	8
NULL	F	W	1
CA	NULL	NULL	2
OR	NULL	NULL	4
TX	NULL	NULL	3
WA	NULL	NULL	9

只有两类不同组合对应的汇总：

- 不同的 StudState 值（不管 StudGender 和 StudMaritalStatus 的值是什么）；
- StudGender 和 StudMaritalStatus 的不同组合（不管 StudState 的值是什么）。

“数据库系统之所以返回这样的结果，是因为我在 GROUPING SETS 列表中指定了两个不同的分组集：StudState；StudGender 和 StudMaritalStatus。如你所见，结果中包含每个州的小计以及每种性别和婚姻状况组合的小计。”

下面来看看如果将一列移出分组集，将其提升为简单分组列，结果将如何。请看下面的请求：

“显示每个州的小计以及每种性别和婚姻状况的小计。”

转换/整理　Select ~~the~~ student state, student gender, student marital status, ~~and the~~ count(*) ~~of rows~~ from ~~the~~ Students ~~table~~, group~~ed~~ by student state ~~and~~ roll~~ed~~ up ~~by~~ student gender ~~and~~ student marital status（在学生表中，按州分组并按性别和婚姻状况累积，从每个分组中选择州、性别、婚姻状况和行数）

SQL
```
SELECT StudState, StudGender, StudMaritalStatus,
  Count(*) AS Number
FROM Students
GROUP BY StudState,
  ROLLUP (StudMaritalStatus, StudGender);
```

结果类似于下表（这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_Count_ROLLUP_Partial）。

StudState	StudGender	StudMaritalStatus	Number
CA	F	W	1
CA	F	NULL	1
CA	M	S	1
CA	M	NULL	1
CA	NULL	NULL	2
OR	F	M	1
OR	F	S	1
OR	F	NULL	2
OR	M	S	2
OR	M	NULL	2
OR	NULL	NULL	4
TX	F	S	2
TX	F	NULL	2
TX	M	S	1
TX	M	NULL	1
TX	NULL	NULL	3
WA	F	D	1
WA	F	M	1
WA	F	S	3
WA	F	NULL	5
WA	M	S	4
WA	M	NULL	4
WA	NULL	NULL	9

如你所见，它按 StudState 分组，并按 StudGender 和 StudMaritalStatus 累积，结果是 3 级汇总：

- StudState、StudGender 和 StudMaritalStatus 的不同组合；
- StudState 和 StudGender 的不同组合（不管 StudMaritalStatus 的值是什么）；
- 不同的 StudState 值（不管 StudGender 和 StudMaritalStatus 的值是什么）。

通过将 StudState 移出 GROUPING SETS，得到的结果与 CH21_Students_State_Gender_MaritalStatus_Count_CUBE_Order1 的结果相同——只是没有总计。

我们再来试试，但这次使用 CUBE 而不是 GROUPING SETS，并将结果排序，以便你更清楚地看出发生了什么。这里跳过请求和转换/整理步骤，直接给出 SQL。

```
SQL         SELECT StudState, StudGender, StudMaritalStatus,
              Count(*) AS Number
            FROM Students
            GROUP BY StudState,
              CUBE (StudMaritalStatus, StudGender)
            ORDER BY StudState, StudGender, StudMaritalStatus
```

结果如下：

StudState	StudGender	StudMaritalStatus	Number
CA	NULL	NULL	2
CA	NULL	S	1
CA	NULL	W	1

（续）

StudState	StudGender	StudMaritalStatus	Number
CA	F	NULL	1
CA	F	W	1
CA	M	NULL	1
CA	M	S	1
OR	NULL	NULL	4
OR	NULL	M	1
OR	NULL	S	3
OR	F	NULL	2
OR	F	M	1
OR	F	S	1
OR	M	NULL	2
OR	M	S	2
TX	NULL	NULL	3
TX	NULL	S	3
TX	F	NULL	2
TX	F	S	2
TX	M	NULL	1
TX	M	S	1
WA	NULL	NULL	9
WA	NULL	D	1
WA	NULL	M	1
WA	NULL	S	7
WA	F	NULL	5
WA	F	D	1
WA	F	M	1
WA	F	S	3
WA	M	NULL	4
WA	M	S	4

这个查询保存在示例数据库 School Scheduling 中，名为 CH21_Students_State_Gender_MaritalStatus_CUBE_Partial。

由于 StudState 在 CUBE 外面，因此不直接参与 CUBE，但你确实获得了每个州的汇总以及每种性别和婚姻状况组合的汇总。请注意，将一列移到外面后，没有返回总计，结果类似于使用 GROUPING SETS 和多个列组合得到的结果。你现在晕了吗？

请大胆使用我提供的示例数据库尝试其他变种和组合。你可以复制一个视图中的 SQL，再不断鼓捣，看看结果如何。

21.7　语句举例

至此，你知道了如何使用分组集编写查询，还知道了可使用它来解决的一些问题类型。我们不再纠缠学生的性别和婚姻状况，来看看其他示例，它们都使用了一种或多种分组集指定方法。这些示例来自各个示例数据库，演示了如何使用分组集以各种方式生成汇总。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库中，名称以 CH20 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

> ❖ **说明**　别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录B。这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。为简化流程，我在下面的示例中将转换和整理合二为一了。

21.7.1　ROLLUP 使用示例

- **Sales Orders 数据库**

“按州显示每类商品的订单数和销售额；另外，显示类别汇总和总计。”

转换/整理

Select CategoryDescription, CustState, ~~the~~ count ~~of~~ DISTINCT orders.OrderNumber, ~~and the~~ sum ~~of~~ (QuotedPrice ~~times~~ * QuantityOrdered) as Price from ~~the~~ Order_Details ~~table~~ inner join~~ed with the~~ Orders ~~table~~ on Orders.OrderNumber = Order_Details.OrderNumber inner join~~ed with the~~ Customers ~~table~~ on Customers.CustomerID = Orders.CustomerID inner join~~ed with the~~ Products ~~table~~ on Products.ProductNumber = Order_Details.ProductNumber inner join~~ed with the~~ Categories ~~table~~ on Categories.CategoryID = Products.CategoryID ~~summarized by~~ GROUP BY ROLLUP (CategoryDescription ~~and~~ CustState)［依次基于 Orders.OrderNumber = Order_Details.OrderNumber 将 Order_Details 内连接到 Orders 表、基于 Customers.CustomerID = Orders.CustomerID 内连接到 Customers 表、基于 Products.ProductNumber = Order_Details.ProductNumber 内连接到 Products 表、基于 Categories.CategoryID = Products.CategoryID 内连接到 Categories 表；按 CategoryDescription 和 CustState 分组累积，并从每个分组中选择 CategoryDescription、CustState、不同 Orders.OrderNumber 的数量以及 QuotedPrice times 与 QuantityOrdered 的乘积总和（作为 Price）］

talSQL

```
SELECT PC.CategoryDescription, C.CustState,
  COUNT(DISTINCT O.OrderNumber) AS OrderCount,
  SUM(OD.QuotedPrice * QuantityOrdered) AS Revenue
FROM Order_Details AS OD
  INNER JOIN Orders AS O
    ON O.OrderNumber = OD.OrderNumber
  INNER JOIN Customers AS C
    ON C.CustomerID = O.CustomerID
  INNER JOIN Products AS P
    ON P.ProductNumber = OD.ProductNumber
  INNER JOIN Categories AS PC
    ON PC.CategoryID = P.CategoryID
GROUP BY ROLLUP
  (PC.CategoryDescription, C.CustState)
```

CH21_ProductCategory_CustomerState_Revenue_ROLLUP（31 行）

CategoryDescription	CustState	OrderCount	Price
Accessories	CA	174	$85,201.52
Accessories	OR	122	$56,551.79
Accessories	TX	112	$89,104.78
Accessories	WA	37	$141,212.93
Accessories	NULL	94	$372,071.02
Bikes	CA	69	$729,481.45
	<< 其他行 >>		
Tires	NULL	257	$25,249.24
Components	NULL	586	$244242.53
NULL	NULL	933	$4,630,731.37

我们来比较一下。如果只是按这两列分组，将得到每种类别描述和州组合的汇总，但没有总计；如果使用 CUBE 和这两列，将这两个列指定为分组列，将得到每种类型描述和州组合的汇总、每种类别的汇总、每个州的汇总以及总计。由于这个问题只要求同时提供类别汇总和总计，因此我使用了 ROLLUP。

- **School Scheduling 数据库**

“显示在接下来的两个学期，每个教室有多少堂课，并按教学楼、教室、学期和科目计算小计，同时显示总计。”

由于 Classes 表包含的上课日期列表不规范，编写解决这个问题的 SQL 有点棘手，因此我将以解决方案 CH20_Class_Schedule_Calendar 为基础，它对于每堂课都返回一行。如果你心存疑惑，可能需要复习一下前一章的相应示例。

转换/整理　Select BuildingCode, ClassRoomID, SemesterNo, SubjectCode, ~~and the count of classes~~ Count(*) from ~~the~~ CH20_Class_Schedule_Calendar ~~view summarized by~~ GROUP BY ROLLUP (BuildingCode, ClassRoomID, SemesterNo, SubjectCode)（将视图 CH20_Class_Schedule_Calendar 按 BuildingCode、ClassRoomID、SemesterNo 和 SubjectCode 分组累积，并从每个分组中选择 BuildingCode、ClassRoomID、SemesterNo、SubjectCode 和行数）

talSQL

```
SELECT BuildingCode, ClassRoomID, SemesterNo,
  SubjectCode, Count(*) AS NumberOfSessions
FROM CH20_Class_Schedule_Calendar
GROUP BY ROLLUP(BuildingCode, ClassRoomID,
  SemesterNo, SubjectCode);
```

CH21_Building_ClassRoom_Semester_Subject_Count_ROLLUP（212 行）

BuildingCode	ClassRoomID	SemesterNo	SubjectCode	NumberOfSessions
AS	1514	1	JRN 104	29
AS	1514	1	NULL	29
AS	1514	2	JRN 104	29
AS	1514	2	NULL	29
AS	1514	NULL	NULL	58
		<< 其他行 >>		
TB	1642	2	CIS 114	58
TB	1642	2	NULL	58
TB	1642	NULL	NULL	117
TB	NULL	NULL	NULL	439
NULL	NULL	NULL	NULL	7221

21.7.2　CUBE 使用示例

- **Bowling League 数据库**

“我要知道每个球队中来自相同城市的投球手的平均原始得分；请提供球队和城市的每种组合的平均得分、每个球队的平均得分、每个城市的平均得分以及所有球员的平均得分。”

转换/整理　Select TeamName, BowlerCity, ~~and the average of~~ Avg(HandicapScore) as AvgHandicap from ~~the~~ Teams ~~table~~ inner join~~ed with the~~ Bowlers ~~table~~ on Bowlers.TeamID = Teams.TeamID inner join~~ed with the~~ Bowler_Scores ~~table~~ on Bowler_Scores.BowlerID = Bowlers.BowlerID ~~summarized in sets by~~ GROUP BY CUBE (TeamName, and BowlerState)［依次基于 Bowlers.TeamID = Teams.TeamID 将 Teams 表内连接到 Bowlers 表、基于 Bowler_Scores.BowlerID = Bowlers.BowlerID 内连接到 Bowler_Scores 表，分别按 TeamName 和 BowlerState 中的零、一和两个分组，从每个分组中选择 TeamName、BowlerCity 和 Avg(HandicapScore)（作为 AvgHandicap）］

SQL

```
SELECT T.TeamName, B.BowlerCity,
  Avg(BS.HandicapScore) AS AvgHandicap
FROM Teams AS T
  INNER JOIN Bowlers AS B
    ON B.TeamID = T.TeamID
  INNER JOIN Bowler_Scores AS BS
    ON BS.BowlerID = B.BowlerID
GROUP BY CUBE (T.TeamName, B.BowlerCity);
```

这里的线索是，这个问题不仅要求球队和城市的每种组合的平均得分，还要求每个城市和每个球队的平均得分，因此需要使用 CUBE。

CH21_Team_City_AverageHandicapScore_CUBE（44 行）

TeamName	BowlerCity	AvgHandicap
Barracudas	Auburn	197
Manatees	Auburn	196
Sharks	Auburn	196
Swordfish	Auburn	193
NULL	Auburn	196
Marlins	Ballard	196
Terrapins	Ballard	195
NULL	Ballard	196
Terrapins	Bellevue	194
	<< 其他行 >>	

- **Sales Orders 数据库**

“对于每种类别的商品，按供应商所在的州显示存货量，同时显示每种类别商品的存货量以及所有商品的存货量。”

转换/整理

Select CategoryDescription, VendState, ~~and the~~ sum ~~of~~ (QuantityOfHand) as QOH from ~~the~~ Products ~~table~~ inner join~~ed with the~~ Categories ~~table~~ on Categories.CategoryID = Products.CategoryID inner join~~ed with the~~ Product_Vendors ~~table~~ on Product_Vendors.ProductNumber = Products.ProductNumber inner join~~ed with the~~ Vendors ~~table~~ on Vendors.VendorID = Product_Vendors.VendorID ~~summarized in sets by~~ GROUP BY CUBE(CategoryDescription, ~~and~~ VendState)［依次基于 Categories.CategoryID = Products.CategoryID 将 Products 表内连接到 Categories 表、基于 Product_Vendors.ProductNumber = Products.ProductNumber 内连接到 Product_Vendors 表、基于 Vendors.VendorID = Product_Vendors.VendorID 内连接到 Vendors 表；分别按 CategoryDescription 和 VendState 中的零、一和两个分组，从每个分组中选择 CategoryDescription、VendState 和 QuantityOfHand 总和（作为 QOH）］

SQL

```
SELECT PC.CategoryDescription, V.VendState,
  SUM(P.QuantityOnHand) AS QOH
FROM Products AS P
   INNER JOIN Categories AS PC
    ON PC.CategoryID = P.CategoryID
  INNER JOIN Product_Vendors AS PV
    ON PV.ProductNumber = P.ProductNumber
  INNER JOIN Vendors AS V
    ON V.VendorID = PV.VendorID
GROUP BY CUBE (PC.CategoryDescription,
      V.VendState)
```

CH21_ProductCategory_VendorState_QOH_CUBE（39 行）

CategoryDescription	VendState	QOH
Accessories	AK	48
Bikes	AK	8
Car racks	AK	14
Clothing	AK	94
Components	AK	278
Tires	AK	60
NULL	AK	502
Accessories	CA	54
NULL	NULL	1914
Accessories	NULL	642
Bikes	NULL	48
Car racks	NULL	28
Clothing	NULL	222
Components	NULL	794
Tires	NULL	180
	<< 其他行 >>	

21.7.3　GROUPING SETS 使用示例

- **Bowling League 数据库**

“显示每个球队的投球手总共参加了多少局比赛，以及每个城市的投球手总共参加了多少局比赛。”

转换/整理　Select TeamName, BowlerCity, ~~and the count~~ count(*) ~~of~~ from ~~the~~ Teams ~~table~~ inner join~~ed with the~~ Bowlers ~~table~~ on Bowlers.TeamID = Teams.TeamID inner join~~ed with the~~ Bowler_Scores ~~table~~ on Bowler_Scores.BowlerID = Bowlers.BowlerID ~~summarized both by~~ GROUP BY GROUPING SETS (TeamName, ~~and~~ BowlerCity)（依次基于 Bowlers.TeamID = Teams.TeamID 将 Teams 内连接到 Bowlers 表、基于 Bowler_Scores.BowlerID = Bowlers.BowlerID 内连接到 Bowler_Scores 表；分别按 TeamName 和 BowlerCity 分组，并从每个分组中选择 TeamName、BowlerCity 和行数）

SQL
```
SELECT T.TeamName, B.BowlerCity, Count(*) AS
  GamesBowled
FROM Teams AS T
  INNER JOIN Bowlers AS B
    ON B.TeamID = T.TeamID
  INNER JOIN Bowler_Scores AS BS
    ON BS.BowlerID = B.BowlerID
GROUP BY GROUPING SETS (T.TeamName,
  B.BowlerCity);
```

请注意，我要的不是球队和城市的每种组合的总局数，而是每个球队的总局数和每个城市的总局数，因此 GROUPING SETS 是理想的选择。

CH21_Team_City_GamesBowled_GROUPING_SETS（18 行）

TeamName	BowlerCity	GamesBowled
NULL	Auburn	210
NULL	Ballard	84
NULL	Bellevue	42
NULL	Bothell	84
NULL	Duvall	126
NULL	Kirkland	126
NULL	Redmond	294
NULL	Seattle	168
NULL	Tacoma	42
NULL	Woodinville	168
Barracudas	NULL	168
Dolphins	NULL	168
Manatees	NULL	168
Marlins	NULL	168
Orcas	NULL	168
Sharks	NULL	168
Swordfish	NULL	168
Terrapins	NULL	168

- **Entertainment Agency 数据库**

“显示喜欢每种音乐风格的顾客数以及邮政编码相同的顾客数。”

转换/整理　Select StyleName, CustZipCode, ~~and the count~~ count(*)~~ of~~ from ~~the~~ Customers ~~table~~ inner join~~ed with the~~ Musical_Preferences ~~table~~ on Musical_Preferences.CustomerID = Customers.CustomerID inner join~~ed with the~~ Musical_Styles ~~table~~ on Musical_Styles.StyleID = Musical_Preferences.StyleID ~~summarized both by~~ GROUP BY GROUPING SETS (StyleName, and CustZipCode)（依次基于 Musical_Preferences.CustomerID = Customers.CustomerID 将 Customers 表内连接到 Musical_Preferences 表、基于 Musical_Styles.StyleID = Musical_Preferences.StyleID 内连接到 Musical_Styles 表；分别按 StyleName 和 CustZipCode 分组，从每个分组中选择 StyleName、CustZipCode 和行数）

talSQL	`SELECT MS.StyleName, C.CustZipCode, Count(*) AS Instances FROM Customers AS C INNER JOIN Musical_Preferences AS MP ON MP.CustomerID = C.CustomerID INNER JOIN Musical_Styles AS MS ON MS.StyleID = MP.StyleID GROUP BY GROUPING SETS (MS.StyleName, C.CustZipCode)`

CH21_Style_CustomerZipCode_Count_GROUPING_SETS（27 行）

StyleName	CustZipCode	Instances
NULL	98002	2
NULL	98006	14
NULL	98033	7
NULL	98052	4
NULL	98105	2
NULL	98115	3
NULL	98413	4
40’s Ballroom Music	NULL	2
60’s Music	NULL	1
70’s Music	NULL	1
80’s Music	NULL	1
	<< 其他行 >>	

21.8 小结

本章首先阐述了为何需要以不同于第 13 章介绍的方式对数据进行分组，并通过一个示例证明了这一点。

接下来介绍了如下三种 GROUP BY 扩展之间的差别：

- ❑ GROUP BY ROLLUP
- ❑ GROUP BY CUBE
- ❑ GROUP BY GROUPING SETS

接着指出了关键字 ROLLUP、CUBE 和 GROUPING SETS 只是第 13 章讨论的 GROUP BY 语法的扩展，因此第 13 章介绍的限制也适用于它们。

然后总结了这些 GROUP BY 扩展为何很有用，并列举了使用它们编写查询的示例。

下一节列出了你可独自解决的多个问题。

21.9 练习

下面列举了一些请求，以及示例数据库中对应查询的名称（根据查询名称可知道该使用 ROLLUP、CUBE 还是 GROUPING SETS）。如果你想做些练习，可将每个请求转换为 SQL，再将其与我存储在示例数据库中的查询进行比较。如果你编写的 SQL 语句与我保存的查询不完全相同，也不用担心，只要结果集相同就行。

- **Bowling League 数据库**

  (1) 显示每个球队的投球手数量、每个城市的投球手数量、每个球队居住在每个城市的投球手数量以及全部球队的投球手总数。

  解决方案见 CH21_Team_City_Count_CUBE（44 行）。

  (2) 显示每个球队的投球手的平均原始得分以及每个城市的投球手的平均原始得分。

  解决方案见 CH21_Team_City_AverageRawScore_GROUPING_SETS（18 行）。

(3) 显示每个球队中居住在每个城市的投球手平均让分得分、每个球队的投球手平均让分得分以及联盟的全部投球手的平均让分得分。

解决方案见 CH21_Team_City_AverageHandicapScore_ROLLUP（34 行）。

- **Entertainment Agency 数据库**

(1) 显示每个城市的演唱组合演奏的音乐风格总数、每个城市演奏每种音乐风格的演唱组合数、所有演唱组合演奏的音乐风格总数。

解决方案见 CH21_EntertainerCity_Style_ROLLUP（36 行）。

(2) 显示每个城市的顾客喜欢的音乐风格总数、喜欢每种音乐风格的顾客总数、每个城市喜欢每种音乐风格的顾客数。

解决方案见 CH21_CustomerCity_Style_GROUPING_SETS（18 行）。

(3) 对现有的全部演出合约进行分析，指出演出合约和费用在经纪人所在城市、顾客所在城市以及顾客和经纪人所在城市的分布情况。

解决方案见 CH21_AgentCity_CustomerCity_Count_Charge_GROUPING_SETS（34 行）。

- **Recipes 数据库**

(1) 我想知道我的烹饪书中每种菜品类型都包含多少个菜品，以及总共有多少个菜品。务必列出不包含任何菜品的菜品类型。

解决方案见 CH21_RecipeClass_Recipe_Counts_ROLLUP（8 行）。

(2) 我想知道 RecipeClasses 和 IngredientClasses 之间的关系：对于每种菜品类型，指出它使用了多少种食材类型；对于每种食材类型，指出它被用来制作多少种菜品类型。

解决方案见 CH21_RecipeClass_IngredientClass_Counts_GROUPING_SETS（25 行）。

(3) 我想更深入地了解 RecipeClasses 和 IngredientClasses 之间的关系：对于菜品类型和食材类型的每种组合，指出它涉及多少种菜品；对于每种食材类型，指出它被用来制作多少种菜品（不管其菜品类型如何）；每种菜品类型包含多少种菜品；总共有多少种菜品。

解决方案见 CH21_RecipeClass_IngredientClass_CUBE（61 行）。

- **Sales Orders 数据库**

(1) 指出每个州的顾客购买的每种商品类型的价格总额、所有顾客购买的每种商品类型的价格总额以及所有顾客购买的所有商品的价格总额。

解决方案见 CH21_ProductCategory_CustomerState_Revenue_CUBE（35 行）。

(2) 显示每个州的供应商提供的每种商品类型的库存量、每种商品类型的库存量以及所有商品的库存总量。

解决方案见 CH21_ProductCategory_VendorState_QOH_ROLLUP（33 行）。

(3) 我想知道每个州的供应商提供的每类商品的数量、每个州的供应商提供的全部商品的数量以及所有供应商提供的全部商品的数量。

请注意，这不会返回销售出去的商品总数。

解决方案见 CH21_VendorState_Category_Count_ROLLUP（43 行）。

- **School Scheduling 数据库**

(1) 告诉我每个学期有多少堂课、每栋教学楼的每个教室有多少堂课、每个科目有多少堂课。

解决方案见 CH21_Semester_Building_ClassRoom_Subject_Count_GROUPING_SETS（82 行）。

(2) 指出每个系提供了多少课程，其中由教授、副教授和讲师授课的课程分别有多少；另外，告诉我整个学校提供多少课程。

请注意，返回的课程总数将超过学校提供的课程数，因为有些课程由多名教员授课。

解决方案见 CH21_Department_Title_Count_ROLLUP（20 行）。

(3) 我想知道学生的选课情况。告诉我全部学生（及每个专业的学生）修完、刚注册和退课的课程数，以及每个专业的学生已修完、刚注册或退课的课程总数。不用按学期划分。

解决方案见 CH21_Major_ClassStatus_Count_GROUPING_SETS（26 行）。

第 22 章

将数据划分到窗口中

“我知道我脑子里有几扇窗户，但以各种方式向世界打开的总是那一扇。”

——塞缪尔·贝克特

本章涵盖如下主题：

- 将数据划分到窗口中有何用
- 计算行号
- 数据排名
- 将数据划分到五分位区间中
- 结合使用窗口和聚合函数
- 语句举例
- 小结
- 练习

本书前面介绍了多种对数据进行分组和聚合的方法，但还没有面面俱到。

在早期的 SQL 标准中，根本没有处理这种数据的功能：结果取决于相邻行。那时大家都觉得行的排列顺序无关紧要，只要能够将筛选器与行匹配就行。虽然能够使用 ORDER BY 对数据进行排序，但它只能用于显示数据，而不能操作数据。这意味着仅使用 SQL 很难（乃至无法）执行诸如生成移动总计（一行显示的总计依赖于它前面各行的值）等运算。

但 SQL:2003 标准推出后，情况发生了变化。SQL:2003 标准引入了窗口函数的概念；窗口函数应用于窗口描述符定义的一系列行，并为每行返回单个值。

本章介绍窗口函数。如果你使用的是苹果计算机，也不用担心：窗口函数与 PC 操作系统一点关系都没有。实际上，在支持窗口函数方面，Microsoft 是后来者，它在 SQL Server 2017 中才引入窗口函数，而 Microsoft Access 至今没引入窗口函数。

> ❖ **说明** 所有示例语句和解决方案都可在相应的示例数据库——Sales Orders、Entertainment Agency、School Scheduling 和 Bowling League 中找到。由于 Microsoft Access 和 MySQL 都不支持窗口函数，因此只有 Microsoft SQL Server 和 PostgreSQL 示例数据库中有解决方案。

22.1 将数据划分到窗口中有何用

在本书前面所有的数据聚合示例中，都用一个小计行替换了被聚合的数据，这让你无法了解这些数据的详情。例如，如果你想知道每位顾客喜欢多少种音乐风格，可能编写类似于下面的查询：

```
SQL      SELECT CustomerID, C.CustFirstName || ' ' || C.CustLastName
           AS Customer,
           COUNT(*) AS Preferences
         FROM Customers AS C
           INNER JOIN Musical_Preferences AS MP
             ON MP.CustomerID = C.CustomerID
         GROUP BY C.CustFirstName, C.CustLastName;
```

结果将类似于下表（这个查询保存到了示例数据库 Entertainment Agency 中，名为 CH22_Customers_PreferredStyles_Count）。

CustomerID	Customer	Preferences
10001	Doris Hartwig	2
10002	Deb Waldal	2
10003	Peter Brehm	2
10004	Dean McCrae	2
10005	Elizabeth Hallmark	2
10006	Matt Berg	2
10007	Liz Keyser	3
10008	Darren Gehring	2
10009	Sarah Thompson	3
10010	Zachary Ehrlich	3
10011	Joyce Bonnicksen	3
10012	Kerry Patterson	2
10013	Estella Pundt	2
10014	Mark Rosales	3
10015	Carol Viescas	3

这很好，确实能够让你准确地知道每位顾客喜欢多少种音乐风格，但并没有指出是哪些音乐风格。

如果能够使用一个窗口函数确定顾客喜欢的音乐风格数，同时返回他喜欢的每种音乐风格，那该多好！这样，你将获得类似于下面的结果：

CustomerID	Customer	StyleName	Preferences
10001	Doris Hartwig	Contemporary	2
10001	Doris Hartwig	Top 40 Hits	2
10002	Deb Waldal	60's Music	2
10002	Deb Waldal	Classic Rock & Roll	2
10003	Peter Brehm	Motown	2
10003	Peter Brehm	Rhythm and Blues	2
10004	Dean McCrae	Jazz	2
10004	Dean McCrae	Standards	2
10005	Elizabeth Hallmark	Classical	2
10005	Elizabeth Hallmark	Chamber Music	2
10006	Matt Berg	Folk	2
10006	Matt Berg	Variety	2
10007	Liz Keyser	70's Music	3
10007	Liz Keyser	Classic Rock & Roll	3
10007	Liz Keyser	Rhythm and Blues	3
10008	Darren Gehring	Contemporary	2
10008	Darren Gehring	Standards	2
10009	Sarah Thompson	Country	3

（续）

CustomerID	Customer	StyleName	Preferences
10009	Sarah Thompson	Country Rock	3
10009	Sarah Thompson	Modern Rock	3
10010	Zachary Ehrlich	Jazz	3
10010	Zachary Ehrlich	Rhythm and Blues	3
10010	Zachary Ehrlich	Salsa	3
10011	Joyce Bonnicksen	40's Ballroom Music	3
10011	Joyce Bonnicksen	Classical	3
10011	Joyce Bonnicksen	Standards	3
10012	Kerry Patterson	Contemporary	2
10012	Kerry Patterson	Show Tunes	2
10013	Estella Pundt	Jazz	2
10013	Estella Pundt	Salsa	2
10014	Mark Rosales	80's Music	3
10014	Mark Rosales	Modern Rock	3
10014	Mark Rosales	Top 40 Hits	3
10015	Carol Viescas	40's Ballroom Music	3
10015	Carol Viescas	Show Tunes	3
10015	Carol Viescas	Standards	3

❖ **说明** 如果你思维敏锐，可能会说“我早就知道如何获得这样的结果——只需使用类似于下面的子查询”：

```
SELECT CustomerID, CustFirstName || ' ' || CustLastName,
   StyleName, (SELECT COUNT(*)
               FROM Musical_Preferences AS MP
               WHERE MP.CustomerID = Customers.
                  CustomerID)
             AS Preferences
FROM Customers INNER JOIN Musical_Preferences
  ON Customers.CustomerID = Musical_Preferences.
     CustomerID
INNER JOIN Musical_Styles
  ON Musical_Styles.StyleID = Musical_Preferences.
     StyleID;
```

你说得没错，但本章的重点是介绍另一种获取这种结果的方法，其效率可能更高。

现在你不仅知道 Doris Hartwig 喜欢两种音乐风格，还知道她喜欢的两种音乐风格是 Contemporary 和 Top 40 Hits。你肯定明白，这样的结果有用得多。

那么如何获得这样的结果呢？你猜到了，使用窗口函数。我们言归正传，介绍如何做吧。

语法

我们来仔细研究一下窗口函数的语法。图 22-1 显示了基本语法图。请注意，窗口函数只能在 SELECT 和 ORDER BY 子句中使用。看起来非常复杂，不是吗？

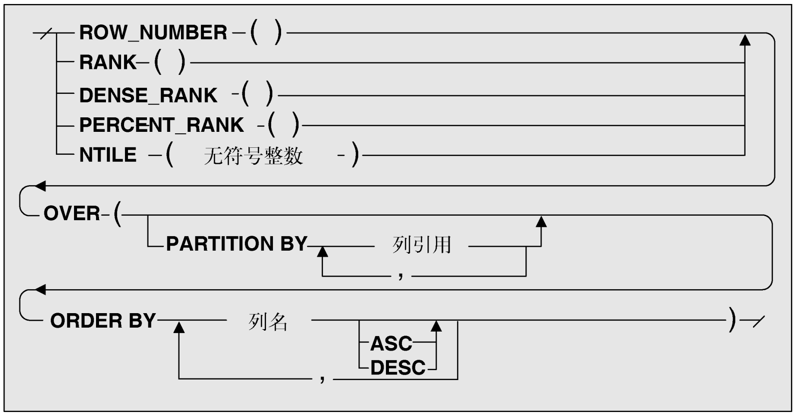


图 22-1　主要窗口函数的语法

但这还不是全部（有点像电视游戏节目）！可将聚合函数转换为窗口，以增大其威力，如图 22-2 所示。

❖ **说明**　图 22-1 和图 22-2 中的列表很长，可以想见窗口函数是一个庞大的主题，仅用一章的篇幅无法全面介绍。因此，我不会介绍所有的窗口函数，而只介绍那些我认为最有用的，但附录 A 介绍了所有窗口函数的语法。

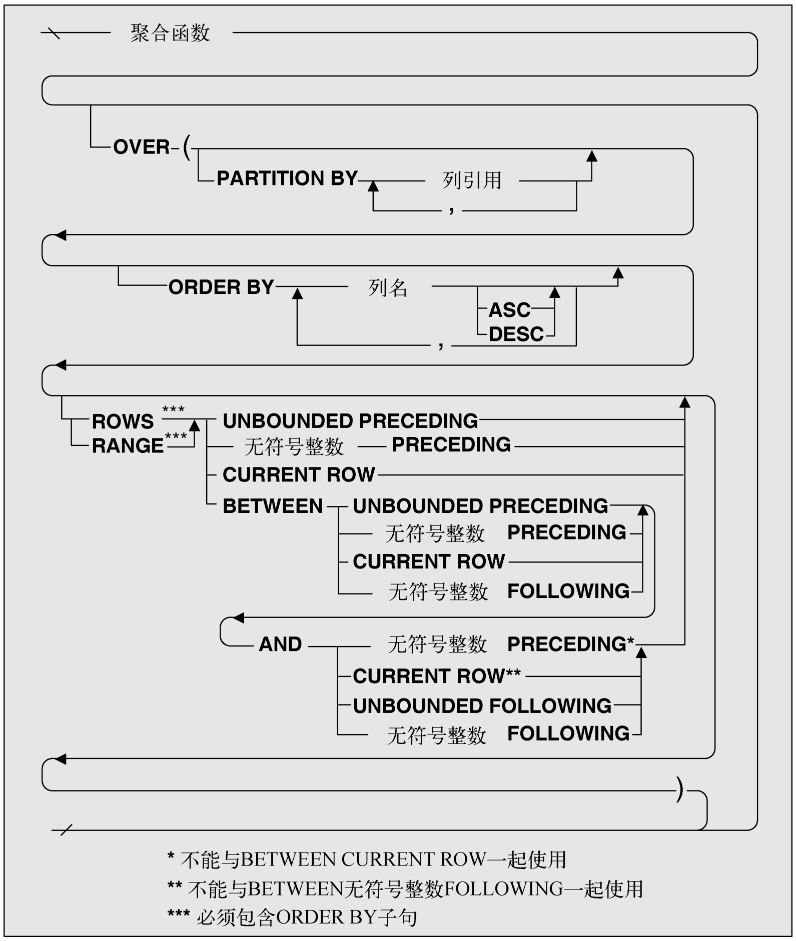


图 22-2　使用聚合函数时的扩展语法

如你所见，这两个图中包含第 12 章介绍的聚合函数（COUNT、SUM、AVG、MIN 和 MAX），还有几个新的函数：ROW_NUMBER()、RANK()、DENSE_RANK()、PERCENT_RANK()和 NTILE()，但看起来还可以指定很多其他的内容。

让你能够使用窗口函数的子句是 OVER()，利用它能够定义聚合函数的应用范围。

GROUP BY 和 OVER()的主要差别在于，GROUP BY 对整个查询应用聚合并将非聚合列合并，因此通常会减少返回的行数，而使用 OVER()时，返回的行数与低层查询相同。对于 OVER()子句指定的范围内的每一行，都将为其返回所有的聚合结果。

在 OVER()子句中，可使用多个谓词：

- ❑ PARTITION BY
- ❑ ORDER BY
- ❑ ROWS（或 RANGE）

谓词 PARTITION BY 指定该如何划分窗口。在前面的示例中，我按顾客划分窗口。在返回前述表的 SQL 中，包含一个 Preferences 列，如下所示。

SQL
```
SELECT C.CustomerID,
  C.CustFirstName || ' ' || C.CustLastName AS Customer,
  MS.StyleName,
  COUNT(*) OVER (
    PARTITION BY C.CustomerID
  ) AS Preferences
FROM Customers AS C
  INNER JOIN Musical_Preferences AS MP
    ON MP.CustomerID = C.CustomerID
  INNER JOIN Musical_Styles AS MS
    ON MS.StyleID = MP.StyleID;
```

现在你知道了如何生成本章开头那样的神奇的表。这个查询保存到了示例数据库 Entertainment Agency 中，名为 CH22_Customers_ PreferredStyles_Details_Count。

如果你没有使用谓词 PARTITION BY，数据库系统将在整个结果集中应用聚合函数。

❖ **说明** 贯穿本章都将使用第 4 章介绍的“请求/转换/整理/SQL”流程。考虑到你现在对这个流程已非常熟悉，为简化流程，我在下面的示例中都将转换和整理合二为一了。

“对于每位顾客，显示他喜欢的音乐风格；同时显示当前所有顾客喜欢的音乐风格数的移动总计。”

转换/整理　Select ~~the customer~~ CustomerID, CustFirstName || ' ' || CustLastName, StyleName, ~~and the~~ count(*) OVER ~~with no partition but~~ order~~ed~~ by CustomerID from ~~the~~ Customers ~~table,~~ inner join~~ed with the~~ Musical_Preferences ~~table~~ on Musical_Preferences.CustomerID = Customers.Customer ID inner join~~ed with the~~ Musical_Styles ~~table~~ on Musical_Styles.StyleID = Musical_Preferences.StyleID（依次基于 Musical_Preferences.CustomerID = Customers.CustomerID 将 Customers 内连接到 Musical_Preferences 表、基于 Musical_Styles.StyleID = Musical_Preferences.StyleID 内连接到 Musical_Styles 表；不分区但按 CustomerID 排序；选择 CustomerID、CustFirstName || ' ' || CustLastName、StyleName 和 count(*)）

SQL
```
SELECT C.CustomerID,
  C.CustFirstName || ' ' || C.CustLastName AS
Customer,
  MS.StyleName,
  COUNT(*) OVER (
    ORDER BY C.CustomerID
  ) AS Preferences
FROM Customers AS C
  INNER JOIN Musical_Preferences AS MP
    ON MP.CustomerID = C.CustomerID
  INNER JOIN Musical_Styles AS MS
    ON MS.StyleID = MP.StyleID;
```

这个查询返回的结果类似于下表（我将这个查询保存到了示例数据库 Entertainment Agency 中，并将其命名为 CH22_Customers_PreferredStyles_Details_NO_PARTITION）。

CustomerID	Customer	StyleName	Preferences
10001	Doris Hartwig	Contemporary	2
10001	Doris Hartwig	Top 40 Hits	2
10002	Deb Waldal	60’s Music	4
10002	Deb Waldal	Classic Rock & Roll	4
10003	Peter Brehm	Motown	6
10003	Peter Brehm	Rhythm and Blues	6
10004	Dean McCrae	Jazz	8
10004	Dean McCrae	Standards	8
10005	Elizabeth Hallmark	Classical	10
10005	Elizabeth Hallmark	Chamber Music	10
10006	Matt Berg	Folk	12
10006	Matt Berg	Variety	12
10007	Liz Keyser	70’s Music	15
10007	Liz Keyser	Classic Rock & Roll	15
10007	Liz Keyser	Rhythm and Blues	15
10008	Darren Gehring	Contemporary	17
10008	Darren Gehring	Standards	17
10009	Sarah Thompson	Country	20
10009	Sarah Thompson	Country Rock	20
10009	Sarah Thompson	Modern Rock	20
10010	Zachary Ehrlich	Jazz	23
10010	Zachary Ehrlich	Rhythm and Blues	23
10010	Zachary Ehrlich	Salsa	23
10011	Joyce Bonnicksen	40’s Ballroom Music	26
10011	Joyce Bonnicksen	Classical	26
10011	Joyce Bonnicksen	Standards	26
10012	Kerry Patterson	Contemporary	28
10012	Kerry Patterson	Show Tunes	28
10013	Estella Pundt	Jazz	30
10013	Estella Pundt	Salsa	30
10014	Mark Rosales	80’s Music	33
10014	Mark Rosales	Modern Rock	33
10014	Mark Rosales	Top 40 Hits	33
10015	Carol Viescas	40’s Ballroom Music	36
10015	Carol Viescas	Show Tunes	36
10015	Carol Viescas	Standards	36

结果与以前相同，但总计不同：不再是每位顾客喜欢的音乐风格数，而是移动总计。在 Doris Hartwig 的两行中，总计还是 2——她喜欢的音乐风格数，但 Deb Waldal 的总计为 4（她自己喜欢的两种音乐风格加上 Doris 喜欢的两种音乐风格），Peter Brehm 的总计为 6（他自己喜欢的两种音乐风格加上 Deb 喜欢的两种音乐风格以及 Doris 喜欢的两种），以此类推。大致而言，我生成了顾客喜欢的音乐风格数的移动总计。

请注意，可以给不同的聚合函数指定不同的 OVER()子句。例如，要获得每位顾客喜欢的音乐风格数以及所有顾客喜欢的音乐风格总数，可使用类似于下面的查询：

“对于每位顾客，指出他喜欢的音乐风格；同时显示每位顾客喜欢的音乐风格数以及所有顾客喜欢的音乐风格数的移动总计。”

转换/整理

Select ~~the customer~~ CustomerID, CustFirstName || ' ' || CustLastName, StyleName, ~~the~~ count(*) OVER partition by CustomerID order~~ed~~ by CustomerID ~~and the~~ count(*) OVER ~~with no partition but~~ order~~ed~~ by CustomerID from ~~the~~ Customers ~~table~~, inner join~~ed with the~~ Musical_Preferences ~~table~~ on Musical_Preferences.CustomerID = Customers.Customer ID inner join~~ed with the~~ Musical_Styles ~~table~~ on Musical_Styles.StyleID = Musical_Preferences.StyleID（依次基于 Musical_Preferences.CustomerID = Customers.CustomerID 将 Customers 表内连接到 Musical_Preferences 表、基于 Musical_Styles.StyleID = Musical_Preferences.StyleID 内连接到 Musical_Styles 表。选择 CustomerID、CustFirstName || ' ' || CustLastName、StyleName、当前顾客喜欢的音乐风格数和当前所有顾客喜欢的音乐风格总数，其中最后两列是分别这样计算得到的：根据 CustomerID 分区并排序，并计算行数；不分区但按 CustomerID 排序，并计算行数）

SQL

```
SELECT C.CustomerID,
  C.CustFirstName || ' ' || C.CustLastName AS
    Customer,
  MS.StyleName,
  COUNT(*) OVER (
    PARTITION BY C.CustomerID
    ORDER BY C.CustomerID
  ) AS CustomerPreferences,
  COUNT(*) OVER (
    ORDER BY C.CustomerID
  ) AS TotalPreferences
FROM Customers AS C
  INNER JOIN Musical_Preferences AS MP
    ON MP.CustomerID = C.CustomerID
  INNER JOIN Musical_Styles AS MS
    ON MS.StyleID = MP.StyleID;
```

这将返回类似于下表的结果（我将这个查询保存到了示例数据库 Entertainment Agency 中，并将其命名为 CH22_Customers_PreferredStyles_Details_Multiple_Counts）。

CustomerID	Customer	StyleName	Customer Preferences	Total Preferences
10001	Doris Hartwig	Contemporary	2	2
10001	Doris Hartwig	Top 40 Hits	2	2
10002	Deb Waldal	60's Music	2	4
10002	Deb Waldal	Classic Rock & Roll	2	4
10003	Peter Brehm	Motown	2	6
10003	Peter Brehm	Rhythm and Blues	2	6
10004	Dean McCrae	Jazz	2	8
10004	Dean McCrae	Standards	2	8
10005	Elizabeth Hallmark	Classical	2	10
10005	Elizabeth Hallmark	Chamber Music	2	10
10006	Matt Berg	Folk	2	12
10006	Matt Berg	Variety	2	12
10007	Liz Keyser	70's Music	3	15
10007	Liz Keyser	Classic Rock & Roll	3	15
10007	Liz Keyser	Rhythm and Blues	3	15
10008	Darren Gehring	Contemporary	2	17
10008	Darren Gehring	Standards	2	17
10009	Sarah Thompson	Country	3	20
10009	Sarah Thompson	Country Rock	3	20
10009	Sarah Thompson	Modern Rock	3	20
10010	Zachary Ehrlich	Jazz	3	23
10010	Zachary Ehrlich	Rhythm and Blues	3	23
10010	Zachary Ehrlich	Salsa	3	23
10011	Joyce Bonnicksen	40's Ballroom Music	3	26
10011	Joyce Bonnicksen	Classical	3	26

（续）

CustomerID	Customer	StyleName	Customer Preferences	Total Preferences
10011	Joyce Bonnicksen	Standards	3	26
10012	Kerry Patterson	Contemporary	2	28
10012	Kerry Patterson	Show Tunes	2	28
10013	Estella Pundt	Jazz	2	30
10013	Estella Pundt	Salsa	2	30
10014	Mark Rosales	80’s Music	3	33
10014	Mark Rosales	Modern Rock	3	33
10014	Mark Rosales	Top 40 Hits	3	33
10015	Carol Viescas	40’s Ballroom Music	3	36
10015	Carol Viescas	Show Tunes	3	36
10015	Carol Viescas	Standards	3	36

请注意，这提供了每位顾客喜欢的音乐风格数，还有当前所有顾客喜欢的音乐风格数移动总计。顾名思义，谓词 ORDER BY 决定了行的返回顺序。可像下面这样修改前一个查询中的谓词 ORDER BY：

“对于每位顾客，指出他喜欢的音乐风格；同时显示每位顾客喜欢的音乐风格数以及所有顾客喜欢的音乐风格数移动总计；根据姓名排序。”

转换/整理

Select ~~the~~ CustomerID, CustFirstName || ' ' || CustLastName, StyleName, ~~the~~ count(*) OVER partition by CustomerID order~~ed~~ by CustLastName, CustFirstName ~~and the~~ count(*) OVER ~~with no partition but~~ order~~ed~~ by CustLastName, CustFirstName from ~~the~~ Customers ~~table~~, inner join~~ed with the~~ Musical_Preferences ~~table~~ on Musical_Preferences.CustomerID = Customers.CustomerID inner join~~ed with the~~ Musical_Styles ~~table~~ on Musical_Styles.StyleID = Musical_Preferences.StyleID（依次基于 Musical_Preferences.CustomerID = Customers.CustomerID 将 Customers 表内连接到 Musical_Preferences 表、基于 Musical_Styles.StyleID = Musical_Preferences.StyleID 内连接到 Musical_Styles 表。选择 CustomerID、CustFirstName || ' ' || CustLastName、StyleName、当前顾客喜欢的音乐风格数和当前所有顾客喜欢的音乐风格总数，其中最后两列是分别这样计算得到的：根据 CustomerID 分区、按 CustLastName 和 CustFirstName 排序并计算行数；不分区但按 CustLastName 和 CustFirstName 排序，并计算行数）

SQL

```
SELECT C.CustomerID,
  C.CustFirstName || ' ' || C.CustLastName
    AS Customer,
  MS.StyleName,
  COUNT(*) OVER (
    PARTITION BY C.CustomerID
      ORDER BY C.CustLastName, C.CustFirstName
  ) AS CustomerPreferences,
  COUNT(*) OVER (
    ORDER BY C.CustLastName, C.CustFirstName
  ) AS TotalPreferences
FROM Customers AS C
  INNER JOIN Musical_Preferences AS MP
    ON MP.CustomerID = C.CustomerID
  INNER JOIN Musical_Styles AS MS
    ON MS.StyleID = MP.StyleID;
```

这将返回类似于下表的结果（我将这个查询保存到了示例数据库 Entertainment Agency 中，并将其命名为 CH22_Customers_PreferredStyles_Details_Multiple_Counts_Sort1）。

CustomerID	Customer	StyleName	Customer Preferences	Total Preferences
10006	Matt Berg	Folk	2	2
10006	Matt Berg	Variety	2	2
10011	Joyce Bonnicksen	40’s Ballroom Music	3	5
10011	Joyce Bonnicksen	Classical	3	5

（续）

CustomerID	Customer	StyleName	Customer Preferences	Total Preferences
10011	Joyce Bonnicksen	Standards	3	5
10003	Peter Brehm	Motown	2	7
10003	Peter Brehm	Rhythm and Blues	2	7
10010	Zachary Ehrlich	Jazz	3	10
10010	Zachary Ehrlich	Rhythm and Blues	3	10
10010	Zachary Ehrlich	Salsa	3	10
10008	Darren Gehring	Contemporary	2	12
10008	Darren Gehring	Standards	2	12
10005	Elizabeth Hallmark	Classical	2	14
10005	Elizabeth Hallmark	Chamber Music	2	14
10001	Doris Hartwig	Contemporary	2	16
10001	Doris Hartwig	Top 40 Hits	2	16
10007	Liz Keyser	70's Music	3	19
10007	Liz Keyser	Classic Rock &	3	19
10007	Liz Keyser	Rhythm and Blues	3	19
10004	Dean McCrae	Jazz	2	21
10004	Dean McCrae	Standards	2	21
10012	Kerry Patterson	Contemporary	2	23
10012	Kerry Patterson	Show Tunes	2	23
10013	Estella Pundt	Jazz	2	25
10013	Estella Pundt	Salsa	2	25
10014	Mark Rosales	80's Music	3	28
10014	Mark Rosales	Modern Rock	3	28
10014	Mark Rosales	Top 40 Hits	3	28
10009	Sarah Thompson	Country	3	31
10009	Sarah Thompson	Country Rock	3	31
10009	Sarah Thompson	Modern Rock	3	31
10015	Carol Viescas	40's Ballroom Music	3	34
10015	Carol Viescas	Show Tunes	3	34
10015	Carol Viescas	Standards	3	34
10002	Deb Waldal	60's Music	2	36
10002	Deb Waldal	Classic Rock & Roll	2	36

如你所见，最终输出现在是按顾客的姓和名而不是顾客 ID 排序的，但请注意，ORDER BY 子句与 PARTITION 子句必须保持一致，否则结果可能会令人迷惑。

“对于每位顾客，指出他喜欢的音乐风格；同时显示每位顾客喜欢的音乐风格数以及所有顾客喜欢的音乐风格数移动总计；按音乐风格名排序。”

转换/整理　Select ~~the customer~~ CustomerID, CustFirstName || ' ' || CustLastName, StyleName, ~~the~~ count(*) OVER partition by CustomerID order~~ed~~ by StyleName ~~and the~~ count(*) OVER ~~with no partition but~~ order~~ed~~ by StyleName from ~~the~~ Customers ~~table,~~ inner join~~ed with the~~ Musical_Preferences ~~table~~ on Musical_Preferences.CustomerID = Customers.Customer ID inner join~~ed with the~~ Musical_Styles ~~table~~ on Musical_Styles.StyleID = Musical_Preferences.StyleID（依次基于 Musical_Preferences.CustomerID = Customers.CustomerID 将 Customers 表内连接到 Musical_Preferences 表、基于 Musical_Styles.StyleID = Musical_Preferences.StyleID 内连接到 Musical_Styles 表。选择 CustomerID、CustFirstName || ' ' || CustLastName、StyleName、当前顾客喜欢的音乐风格数和当前所有顾客喜欢的音乐风格总数，其中最后两列是分别这样计算得到的：根据 CustomerID 分区、按 StyleName 排序并计算行数；不分区但按 StyleName 排序，并计算行数）

SQL
```
SELECT C.CustomerID,
  C.CustFirstName || ' ' || C.CustLastName AS
    Customer,
  MS.StyleName,
  COUNT(*) OVER (
   PARTITION BY C.CustomerID
   ORDER BY MS.StyleName
 ) AS CustomerPreferences,
  COUNT(*) OVER (
    ORDER BY MS.StyleName
  ) AS TotalPreferences
FROM Customers AS C
  INNER JOIN Musical_Preferences AS MP
    ON MP.CustomerID = C.CustomerID
  INNER JOIN Musical_Styles AS MS
    ON MS.StyleID = MP.StyleID;
```

这个查询返回的结果类似于下表（这个查询保存到了示例数据库 Entertainment Agency 中，名为 CH22_Customers_PreferredStyles_Details_Multiple_Counts_Sort2）。

CustomerID	Customer	StyleName	Customer Preferences	Total Preferences
10011	Joyce Bonnicksen	40's Ballroom Music	1	2
10015	Carol Viescas	40's Ballroom Music	1	2
10002	Deb Waldal	60's Music	1	3
10007	Liz Keyser	70's Music	1	4
10014	Mark Rosales	80's Music	1	5
10005	Elizabeth Hallmark	Chamber Music	1	6
10007	Liz Keyser	Classic Rock & Roll	2	8
10002	Deb Waldal	Classic Rock & Roll	2	8
10005	Elizabeth Hallmark	Classical	2	10
10011	Joyce Bonnicksen	Classical	2	10
10012	Kerry Patterson	Contemporary	1	13
10008	Darren Gehring	Contemporary	1	13
10001	Doris Hartwig	Contemporary	1	13
10009	Sarah Thompson	Country	1	14
10009	Sarah Thompson	Country Rock	2	15
10006	Matt Berg	Folk	1	16
10004	Dean McCrae	Jazz	1	19
10010	Zachary Ehrlich	Jazz	1	19
10013	Estella Pundt	Jazz	1	19
10014	Mark Rosales	Modern Rock	2	21
10009	Sarah Thompson	Modern Rock	3	21
10003	Peter Brehm	Motown	1	22
10003	Peter Brehm	Rhythm and Blues	2	25
10007	Liz Keyser	Rhythm and Blues	3	25
10010	Zachary Ehrlich	Rhythm and Blues	2	25
10010	Zachary Ehrlich	Salsa	3	27
10013	Estella Pundt	Salsa	2	27
10015	Carol Viescas	Show Tunes	2	29
10012	Kerry Patterson	Show Tunes	2	29
10011	Joyce Bonnicksen	Standards	3	33
10015	Carol Viescas	Standards	3	33
10008	Darren Gehring	Standards	2	33
10004	Dean McCrae	Standards	2	33

（续）

CustomerID	Customer	StyleName	Customer Preferences	Total Preferences
10001	Doris Hartwig	Top 40 Hits	2	35
10014	Mark Rosales	Top 40 Hits	3	35
10006	Matt Berg	Variety	2	36

这个输出是根据我的要求按风格名排序的。TotalPreferences 看起来好像没问题（两个人喜欢 40's Ballroom Music，因此这两行的 TotalPreferences 都是 2；一个人喜欢 60's Music，因此 TotalPreferences 为 3；TotalPreferences 是移动总计——喜欢前面各种风格的人数加上喜欢当前风格的人数）。CustomerPreferences 理解起来更难些，它是特定顾客喜欢的音乐风格数移动总计。特定顾客的第一行为 1，第二行为 2，以此类推。

❖ **说明** 在 OVER()子句中，可不指定任何可选子句，这将对 FROM 和 WHERE 返回的所有行计算聚合结果，并显示在每行中。例如，下面的查询在 NumRows 列中显示总行数：

```
SELECT C.CustomerID,
  C.CustFirstName || ' ' || C.CustLastName AS Customer,
  MS.StyleName,
  COUNT(*) OVER () AS NumRows
FROM Customers AS C
  INNER JOIN Musical_Preferences AS MP
    ON MP.CustomerID = C.CustomerID
  INNER JOIN Musical_Styles AS MS
    ON MS.StyleID = MP.StyleID;
```

这是一种无须使用 GROUP BY 子句就能显示所有行以及总计的方式。

利用谓词 ROWS（或 RANGE）可以指定相对于当前行的范围，从而进一步限制分区包含的数据行。

利用 ROWS 子句可以指定物理范围，即当前行的前几行或后几行。例如，你可指定 ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING，这意味着当前行以及紧跟在它后面的那一行。

利用 RANGE 可以指定逻辑范围，即值相对于当前行的值在指定范围内的行。

❖ **说明** 使用 ROWS 或 RANGE 子句时，必须同时使用 ORDER BY 子句。如果 ORDER BY 指定了多个列，CURRENT ROW FOR RANGE 将根据所有这些列来确定范围。

我们来看一个例子，看看计算喜欢的音乐风格总数时，使用 ROWS 和 RANGES 有什么不同（别担心，稍后将更详细地介绍如何将诸如 SUM 等聚合函数与 OVER()子句结合起来使用）。我们直接编写 SQL，在其中使用 SUM 与 COUNT OVER ROWS 以及 SUM 与 COUNT OVER RANGE，看看结果有什么不同。

> “对于我们有顾客的每个城市，显示顾客及其喜欢的音乐风格数。另外，显示基于顾客的音乐风格数移动总计以及基于城市的音乐风格数移动总计。”

```
SQL          SELECT C.CustCity,
               C.CustFirstName || ' ' || C.CustLastName AS Customer,
               COUNT(*) AS Preferences,
               SUM(COUNT(*)) OVER (
                 ORDER BY C.CustCity
                   ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
               ) AS TotalUsingRows,
               SUM(COUNT(*)) OVER (
                 ORDER BY C.CustCity
                 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
               ) AS TotalUsingRange
             FROM Customers AS C
               INNER JOIN Musical_Preferences AS MP
                 ON MP.CustomerID = C.CustomerID
             GROUP BY C.CustCity, C.CustFirstName, C.CustLastName;
```

这个查询返回的结果类似于下表（这个查询保存到了示例数据库 Entertainment Agency 中，名为 CH22_Customer_ByCity_PreferredStyles_Sums）。

CustCity	Customer	Preferences	Total UsingRows	Total UsingRange
Auburn	Elizabeth Hallmark	2	2	2
Bellevue	Estella Pundt	2	4	16
Bellevue	Joyce Bonnicksen	3	7	16
Bellevue	Liz Keyser	3	10	16
Bellevue	Mark Rosales	3	13	16
Bellevue	Sarah Thompson	3	16	16
Kirkland	Darren Gehring	2	18	23
Kirkland	Peter Brehm	2	20	23
Kirkland	Zachary Ehrlich	3	23	23
Redmond	Dean McCrae	2	25	27
Redmond	Kerry Patterson	2	27	27
Seattle	Carol Viescas	3	30	32
Seattle	Doris Hartwig	2	32	32
Tacoma	Deb Waldal	2	34	36
Tacoma	Matt Berg	2	36	36

来看其中两个使用 SUM 的计算列。第一个（TotalUsingRows）使用了 ROWS 谓词 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW，而第二个（TotalUsingRange）使用了 RANGE 谓词 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW。

TotalUsingRows 列的值为当前行的 Preferences 列值加上前一行的 TotalUsingRows 列值。换而言之，结合使用聚合函数 SUM 和谓词 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW 时，将生成移动总计。

TotalUsingRange 列的值截然不同（虽然在第一行和最后一行，其值与 TotalUsingRows 列的值相同）。谓词 RANGE 指定的范围是这样的：在谓词 ORDER BY 指定的列（CustCity）中，值与当前行相同的所有行。换而言之，在第一行中，TotalUsingRange 列的值为该行所指城市（Auburn）对应的所有行的 Preferences 列值总和。城市 Bellevue 对应 5 行，这些行的 Preferences 列值分别为 2、3、3、3 和 3。这些值的总和为 14，将这个值加上前一个城市的总和（2），结果为 16，因此在 Bellevue 对应的 5 行中，TotalUsingRange 列的值都为 16。同理，有 3 行与城市 Kirkland 对应，这 3 行的 Preferences 列值总和为 7；将这个值加上前一个城市的总和（16），结果为 23，因此在城市 Kirkland 对应的 3 行中，TotalUsingRange 列的值都为 23。这也可称为移动总计，只是对应同一个城市的所有行的值都相同。换个角度看，在与特定城市对应的最后一行（Auburn, Elizabeth Hallmark、Bellevue, Sarah Thompson、Kirkland, Zachary Ehrlich 等）中，TotalUsingRange 和 TotalUsingRows 列的值相同。

需要指出的一点是，谓词 ROWS（或 RANGE）只能与聚合函数一起使用。图 22-3 总结了相关的规则。

函　　数	OVER 子句	PARTITION BY	ORDER BY	ROWS 或 RANGE
ROW_NUMBER()	必不可少	可选	必不可少	禁用
RANK()	必不可少	可选	必不可少	禁用
DENSE_RANK()	必不可少	可选	必不可少	禁用
PERCENT_RANK()	必不可少	可选	必不可少	禁用
NTILE(n)	必不可少	可选	必不可少	禁用
聚合函数	可选	可选	可选	可选

图 22-3　对各个窗口函数来说，必不可少、可选和禁用的子句

22.2 计算行号

图 22-1 引入的新函数之一是 ROW_NUMBER()。顾名思义，利用它可以给每行指定不同的行号。

“显示顾客的 CustomerID、姓名和所在的州；按字母顺序排列顾客并编号。”

转换/整理

Select ~~the~~ ROW_NUMBER() OVER order~~ed~~ by CustLastName, CustFirstName, ~~the customer and state~~ CustomerID, CustFirstName || ' ' || CustLastName, CustState from ~~the~~ Customers ~~table~~（根据 CustLastName 和 CustFirstName 对 Customers 表进行排序，再选择 ROW_NUMBER()、CustomerID、CustFirstName || ' ' || CustLastName 和 CustState）

SQL

```
SELECT ROW_NUMBER() OVER (
  ORDER BY CustLastName, CustFirstName
) AS RowNumber,
  C.CustomerID,
  C.CustFirstName || ' ' || C.CustLastName
    AS CustomerName,
  C.CustState
FROM Customers AS C;
```

这个查询返回的结果类似于下表（这个查询保存到了示例数据库 Sales Orders 中，名为 CH22_Customers_Numbering）。

RowNumber	CustomerID	CustomerName	CustState
1	1014	Sam Abolrous	CA
2	1020	Joyce Bonnicksen	WA
3	1004	Robert Brown	TX
4	1009	Andrew Cencini	WA
5	1026	Kirk DeGrasse	TX
6	1019	Zachary Ehrlich	CA
7	1015	Darren Gehring	CA
8	1011	Alaina Hallmark	WA
9	1003	Gary Hallmark	WA
10	1010	Angel Kennedy	TX
11	1012	Liz Keyser	WA
12	1005	Dean McCrae	WA
13	1027	Luke Patterson	OR
14	1025	Maria Patterson	TX
15	1008	Neil Patterson	CA
16	1013	Rachel Patterson	CA
17	1021	Estella Pundt	TX
18	1024	Mark Rosales	TX
19	1023	Julia Schnebly	WA
20	1017	Manuela Seidel	OR
21	1007	Mariya Sergienko	OR
22	1018	David Smith	CA
23	1002	William Thompson	WA
24	1028	Jeffrey Tirekicker	WA
25	1022	Caleb Viescas	CA
26	1006	John Viescas	WA
27	1001	Suzanne Viescas	WA
28	1016	Jim Wilson	OR

当然，由于 ROW_NUMBER 是窗口函数，因此我们可使用 OVER 子句以不同的方式进行分区。

“显示顾客的 CustomerID、姓名和所在的州；在每个州内，按字母顺序排列顾客并编号。”

转换/整理　Select ~~the~~ ROW_NUMBER() OVER partition by CustState, order~~ed~~ by CustLastName, CustFirstName, ~~the~~ CustomerID, CustFirstName || ' ' || CustLastName, ~~and~~ CustState from ~~the~~ Customers ~~table~~（根据 CustState 对 Customers 表进行分区；在每个分区中，根据 CustLastName, CustFirstName 进行排序，并选择 ROW_NUMBER()、CustomerID、CustFirstName || ' ' || CustLastName 和 CustState）

SQL

```
SELECT ROW_NUMBER() OVER (
  PARTITION BY CustState
  ORDER BY CustLastName, CustFirstName
) AS RowNumber,
  C.CustomerID,
  C.CustFirstName || ' ' || C.CustLastName AS
    CustomerName,
  C.CustState
FROM Customers AS C;
```

这个查询返回的结果类似于下表（我将这个查询保存到了示例数据库 Sales Orders 中，并将其命名为 CH22_Customers_Numbering_By_State）。

RowNumber	CustomerID	CustomerName	CustState
1	1014	Sam Abolrous	CA
2	1019	Zachary Ehrlich	CA
3	1015	Darren Gehring	CA
4	1008	Neil Patterson	CA
5	1013	Rachel Patterson	CA
6	1018	David Smith	CA
7	1022	Caleb Viescas	CA
1	1027	Luke Patterson	OR
2	1017	Manuela Seidel	OR
3	1007	Mariya Sergienko	OR
4	1016	Jim Wilson	OR
1	1004	Robert Brown	TX
2	1026	Kirk DeGrasse	TX
3	1010	Angel Kennedy	TX
4	1025	Maria Patterson	TX
5	1021	Estella Pundt	TX
6	1024	Mark Rosales	TX
1	1020	Joyce Bonnicksen	WA
2	1009	Andrew Cencini	WA
3	1011	Alaina Hallmark	WA
4	1003	Gary Hallmark	WA
5	1012	Liz Keyser	WA
6	1005	Dean McCrae	WA
7	1023	Julia Schnebly	WA
8	1002	William Thompson	WA
9	1028	Jeffrey Tirekicker	WA
10	1006	John Viescas	WA
11	1001	Suzanne Viescas	WA

由于根据州进行了分区，因此在每个州中，RowNumber 列的起始值都为 1，而行是根据顾客的姓和名排列的。

22.3　数据排名

图 22-1 引入的另一个新函数是 RANK()，顾名思义，利用它可以对数据行排名。

“列出修完了英语课（English）的学生，并按成绩排名。”

转换/整理

Select ~~the~~ SubjectID, StudFirstName, StudLastName, SubjectName, Grade, ~~and~~ RANK() OVER order~~ed~~ by Grade DESC from ~~the~~ Students ~~table~~ inner join~~ed with the~~ Student_Schedules ~~table~~ on Students.StudentID = Student_Schedules.StudentID inner join~~ed with the~~ Classes ~~table~~ on Classes.ClassID = Student_Schedules.ClassID inner join~~ed with the~~ Subjects ~~table~~ on Subjects.SubjectID = Classes.SubjectID where ClassStatus = 2 and CategoryID = ‘ENG’（依次基于 Students.StudentID = Student_Schedules.StudentID 将 Students 内连接到 Student_Schedules 表、基于 Classes.ClassID = Student_Schedules.ClassID 内连接到 Classes 表、基于 Subjects.SubjectID = Classes.SubjectID 内连接到 Subjects 表；选择 ClassStatus = 2 且 CategoryID = ‘ENG’的行，并根据 Grade 降序排列；从这些行中选择 SubjectID、StudFirstName、StudLastName、SubjectName、Grade 和排名（RANK()））

SQL

```
SELECT Su.SubjectID, St.StudFirstName,
  St.StudLastName, Su.SubjectName,
  SS.Grade,
  RANK() OVER (
    ORDER BY SS.Grade DESC
  ) AS Rank
FROM Students AS St
  INNER JOIN Student_Schedules AS SS
    ON SS.StudentID = St.StudentID
  INNER JOIN Classes AS C
    ON C.ClassID = SS.ClassID
  INNER JOIN Subjects AS Su
    ON Su.SubjectID = C.SubjectID
WHERE SS.ClassStatus = 2
  AND Su.CategoryID = 'ENG';
```

这个查询返回的结果类似于下表（这个查询保存在示例数据库 School Scheduling 中，名为 CH22_English_Students_Rank）。

SubjectID	Stud FirstName	Stud LastName	SubjectName	Grade	Rank
37	Scott	Bishop	Composition - Fundamentals	98.07	1
37	Sara	Sheskey	Composition - Fundamentals	97.59	2
37	John	Kennedy	Composition - Fundamentals	93.01	3
37	Brannon	Jones	Composition - Fundamentals	91.66	4
37	Janice	Galvin	Composition - Fundamentals	91.44	5
38	Kendra	Bonnicksen	Composition - Intermediate	88.91	6
37	George	Chavez	Composition - Fundamentals	88.54	7
37	Marianne	Wier	Composition - Fundamentals	87.4	8
37	David	Hamilton	Composition - Fundamentals	86.33	9
37	Steve	Pundt	Composition - Fundamentals	82.58	10
38	Doris	Hartwig	Composition - Intermediate	81.66	11
37	Michael	Viescas	Composition - Fundamentals	77.59	12
38	Elizabeth	Hallmark	Composition - Intermediate	72.88	13
37	Karen	Smith	Composition - Fundamentals	72.05	14

（续）

SubjectID	Stud FirstName	Stud LastName	SubjectName	Grade	Rank
37	Betsy	Stadick	Composition - Fundamentals	71.09	15
37	Kerry	Patterson	Composition - Fundamentals	70	16
38	Sarah	Thompson	Composition - Intermediate	67.6	17
38	Richard	Lum	Composition - Intermediate	67.19	18

这里没有相同的成绩，因此排名越靠后，成绩越差。

在同一个分区中，如果多行的排名数据相同，则它们的排名将相同。例如，在刚才的表中，如果 Brannon Jones 和 Janice Galvin 的 Composition-Fundamentals 成绩都是 91.66，那么他们都是第 4 名，而 Kendra Bonnicksen 还是第 6 名（跳过第 5 名），如下所示。

SubjectID	StudFirstName	StudLastName	SubjectName	Grade	Rank
37	Scott	Bishop	Composition - Fundamentals	98.07	1
37	Sara	Sheskey	Composition - Fundamentals	97.59	2
37	John	Kennedy	Composition - Fundamentals	93.01	3
37	Brannon	Jones	Composition - Fundamentals	91.66	4
37	Janice	Galvin	Composition - Fundamentals	91.66	4
38	Kendra	Bonnicksen	Composition - Intermediate	88.91	6
		<< 其他行 >>			

图 22-1 还包含函数 DENSE_RANK()和 PERCENT_RANK()。这两个函数与 RANK()的差别很简单：DENSE_RANK()返回当前行之前值不同的行数加 1，即不跳过排名值。例如，在刚才的假设中，对于 Kendra Bonnicksen，函数 DENSE_RANK()返回的排名值为 5。函数 DENSE_RANK 返回的排名是连续的，没有空缺。

函数 PERCENT_RANK()返回一个这样的数字：在当前分区中，之前的行数与总行数减 1 的比值。如果分区中没有值相同的行，那么在第一行中，函数 PERCENT_RANK()返回的值为 0，而最后一行的返回值为 1；对于分区中的其他行，函数 PERCENT_RANK 通过将其排名（RANK()返回的值）减 1 除以分区中的总行数减 1 来计算排名值，公式如下：

$$\text{PERCENT_RANK} = \frac{(rk-1)}{(nr-1)}$$

我们来看一个示例。

“根据平均让分得分对联盟中的所有投球手排名；显示函数 RANK()、DENSE_RANK()和 PERCENT_RANK()返回的排名值，以展示它们的不同（别忘了，平均得分被舍入为整数）。”

转换/整理	Select ~~the~~ BowlerID, BowlerName, ROUND(AVG(HandiCapScore), 0), RANK() OVER order~~ed~~ by ROUND(AVG(HandiCapScore), 0) DESC AS Rank, DENSE_RANK() OVER order~~ed~~ by ROUND(AVG(HandiCapScore), 0) DESC AS DenseRank ~~and~~ PERCENT_RANK() OVER order~~ed~~ by ROUND(AVG(HandiCapScore), 0) DESC AS PercentRank from ~~table~~ Bowlers inner join~~ed with the~~ Bowler_Scores ~~table~~ ON Bowler_Scores.BowlerID = Bowlers.BowlerID, group~~ed~~ by BowlerID, BowlerFirstName ~~and~~ BowlerLastName（基于 Bowler_Scores.BowlerID = Bowlers.BowlerID 将 Bowlers 表内连接到 Bowler_Scores 表，并根据 BowlerID、BowlerFirstName 和 BowlerLastName 进行分组；计算每个分组的平均让分得分并舍入为整数，再根据平均让分得分将所有分组排序；选择每个分组的 BowlerID、BowlerName、平均让分得分、RANK()、DENSE_RANK()和 PERCENT_RANK()）

SQL
```
SELECT B.BowlerID,
  B.BowlerFirstName || ' ' || B.BowlerLastName
    AS BowlerName,
  ROUND(AVG(BS.HandiCapScore), 0) AS AvgHandicap,
  RANK () OVER (
    ORDER BY ROUND(AVG(BS.HandiCapScore), 0) DESC)
      AS Rank,
  DENSE_RANK () OVER (
    ORDER BY ROUND(AVG(BS.HandiCapScore), 0) DESC)
      AS DenseRank,
  PERCENT_RANK () OVER (
    ORDER BY ROUND(AVG(BS.HandiCapScore), 0) DESC)
      AS PercentRank
FROM Bowlers AS B
  INNER JOIN Bowler_Scores AS BS
    ON BS.BowlerID = B.BowlerID
GROUP BY B.BowlerID, B.BowlerFirstName,
  B.BowlerLastName;
```

❖ **说明**　我将计算得到的平均分舍入为整数了，因为保龄球联盟都这样做。计算让分数时需要一个整数值。为了得到正确答案，必须在每个 ORDER BY 子句中包含这个舍入表达式。如果你查看我在 Microsoft SQL Server 中使用的代码，将发现我还使用 CAST 将 HandiCapScore 列转换成了数据类型 FLOAT；如果不这样做，这个数据库系统将返回截断（而不是舍入）的平均值。

这个查询返回的结果类似于下表（这个查询保存在示例数据库 Bowling League 中，名为 CH22_Bowlers_Average_Score_Rankings）。

BowlerID	BowlerName	Avg Handicap	Rank	Dense Rank	Percent Rank
15	Kathryn Patterson	198	1	1	0
6	Neil Patterson	198	1	1	0
27	William Thompson	198	1	1	0
19	John Viescas	198	1	1	0
25	Megan Patterson	197	5	2	0.129032258064516
3	John Kennedy	197	5	2	0.129032258064516
29	Bailey Hallmark	197	5	2	0.129032258064516
14	Gary Hallmark	197	5	2	0.129032258064516
2	David Fournier	196	9	3	0.258064516129032
31	Ben Clothier	196	9	3	0.258064516129032
11	Angel Kennedy	196	9	3	0.258064516129032
26	Mary Thompson	196	9	3	0.258064516129032
7	David Viescas	196	9	3	0.258064516129032
1	Barbara Fournier	196	9	3	0.258064516129032
24	Sarah Thompson	196	9	3	0.258064516129032
18	Michael Hernandez	195	16	4	0.483870967741936
10	Doug Steele	195	16	4	0.483870967741936
12	Carol Viescas	195	16	4	0.483870967741936
9	Alastair Black	195	16	4	0.483870967741936
5	Ann Patterson	195	16	4	0.483870967741936
22	Alaina Hallmark	195	16	4	0.483870967741936
16	Richard Sheskey	194	22	5	0.67741935483871
20	Suzanne Viescas	194	22	5	0.67741935483871
28	Michael Viescas	194	22	5	0.67741935483871
23	Caleb Viescas	194	22	5	0.67741935483871

（续）

BowlerID	BowlerName	Avg Handicap	Rank	Dense Rank	Percent Rank
21	Zachary Ehrlich	194	22	5	0.67741935483871
4	Sara Sheskey	194	22	5	0.67741935483871
13	Elizabeth Hallmark	194	22	5	0.67741935483871
30	Rachel Patterson	194	22	5	0.67741935483871
32	Joe Rosales	193	30	6	0.935483870967742
8	Stephanie Viescas	193	30	6	0.935483870967742
17	Kendra Hernandez	193	30	6	0.935483870967742

如你所见，Rank 和 DenseRank 列的值不同。前 4 位投球手（Kathryn Patterson、Neil Patterson、William Thompson 和 John Viescas）的平均得分一样高，都是 198，因此这两个函数返回的排名都是 1。接下来的 4 位投球手（Megan Patterson、John Kennedy、Bailey Hallmark 和 Gary Hallmark）的平均得分也相同，都是 197，因此无论是在 Rank 列还是 DenseRank 列中，他们的值都相同。由于排名在他们前面的投球手有 4 位，因此函数 RANK()返回 5——排在他们前面的投球手数加 1；然而，在他们前面的不同平均得分只有一个（198），因此函数 DENSE_RANK()返回 2——之前的不同平均得分数量加 1。

继续按这样的方式计算排名值。接下来的 7 位投球手（David Fournier、Ben Clothier、Angel Kennedy、Mary Thompson、David Viescas、Barbara Fournier 和 Sarah Thompson）的平均得分相同，都是 196。由于排在他们前面的投球手有 8 位，因此对于这些投球手，函数 RANK()返回的值都是 9；然而，之前的不同值只有两个（198 和 197），因此函数 DENSE_RANK()返回 3。最后，对于 Joe Rosales、Stephanie Viescas 和 Kendra Hernandez，函数 RANK()返回的值都是 30（因为排在他们前面的投球手有 29 位），而 DENSE_RANK()返回的值为 6，因为之前有 5 个不同的值。

再来看看 PercentRank 列。由于 Kathryn Patterson、Neil Patterson、William Thompson 和 John Viescas 的平均得分是最高的，因此在表示他们的行中，函数 PERCENT_RANK()返回的值为 0。接下来的 4 位投球手（Megan Patterson、John Kennedy、Bailey Hallmark 和 Gary Hallmark）的排名（RANK()返回的值）都是 5，而分区包含的行数为 32，根据前面的公式，这意味着函数 PERCENT_RANK 返回的值如下：

$$\frac{(rk-1)}{(nr-1)}=\frac{(5-1)}{(32-1)}=0.129032258064516$$

同理，接下来的 7 位投球手（David Fournier、Ben Clothier、Angel Kennedy、Mary Thompson、David Viescas、Barbara Fournier 和 Sarah Thompson）的排名都是 9，因此函数 PERCENT_RANK 返回的值如下：

$$\frac{(rk-1)}{(nr-1)}=\frac{(9-1)}{(32-1)}=0.258064516129032$$

这个示例没有展示的一点是，如果排在最后的只有一位投球手，那么对于这位投球手，函数 PERCENT_RANK 将返回 1。由于 Joe Rosales、Stephanie Viescas 和 Kendra Hernandez 并列排名倒数第一，因此对于这些投球手，函数 PERCENT_RANK 都返回如下值：

$$\frac{(rk-1)}{(nr-1)}=\frac{(30-1)}{(32-1)}=0.935483870967742$$

22.4 将数据划分到五分位区间中

你可能还记得，第 20 章介绍了如何计算五分位区间：交叉连接两个查询（一个计算每个修完了英语课的学生的排名，另一个计算修完了英语课的学生总数）。通过使用窗口函数，可避免这种繁重的手工数据检索工作。事实上，前面随窗口函数一起引入了聚合函数 NTILE，利用它可以将数据划分为任意数量的区间。

"列出修完了英语课的学生，按成绩排名，并指出每个学生位于哪个五分区间中。"

转换/整理	Select ~~the~~ SubjectID, StudFirstName, StudLastName, SubjectName, Grade, RANK() OVER order~~ed~~ by Grade DESC ~~and~~ NTILE (5) OVER ordered by Grade DESC from ~~the~~ Students ~~table~~ inner join~~ed with the~~ Student_Schedules ~~table~~ on Students.StudentID = Student_Schedules.StudentID inner join~~ed with the~~ Classes ~~table~~ on Classes.ClassID = Student_Schedules.ClassID inner join~~ed with the~~ Subjects ~~table~~ on Subjects.SubjectID = Classes. SubjectID where ClassStatus = 2 and CategoryID = 'ENG'（依次基于 Students.StudentID = Student_Schedules.StudentID 将 Students 表内连接到 Student_Schedules 表、基于 Classes.ClassID = Student_Schedules.ClassID 内连接到 Classes 表、基于 Subjects.SubjectID = Classes.SubjectID 内连接到 Subjects 表；选择 ClassStatus = 2 且 CategoryID = 'ENG'的行，并按 Grade 降序排列它们；从每行中选择 SubjectID、StudFirstName、StudLastName、SubjectName、RANK()和 NTILE (5)）

SQL

```
SELECT Su.SubjectID, St.StudFirstName,
  St.StudLastName,
  SS.ClassStatus,
  SS.Grade, Su.CategoryID,
  Su.SubjectName,
  RANK() OVER (ORDER BY Grade DESC) AS Rank,
  NTILE(5) OVER(ORDER BY Grade DESC) AS Quintile
FROM Subjects AS Su
  INNER JOIN Classes AS C
    ON C.SubjectID = S.SubjectID
  INNER JOIN Student_Schedules AS SS
    ON SS.ClassID = C.ClassID)
  INNER JOIN Students AS St
    ON St.StudentID = SS.StudentID
WHERE SS.ClassStatus = 2
  AND Su.CategoryID = 'ENG';
```

这个查询返回的结果如下表所示（这个查询保存在示例数据库 School Scheduling 中，名为 CH22_English_Students_Quintiles）。

Subject ID	Stud FirstName	Stud LastName	Subject Name	Grade	Rank	Quintile
37	Scott	Bishop	Composition - Fundamentals	98.07	1	1
37	Sara	Sheskey	Composition - Fundamentals	97.59	2	1
37	John	Kennedy	Composition - Fundamentals	93.01	3	1
37	Brannon	Jones	Composition - Fundamentals	91.66	4	1
37	Janice	Galvin	Composition - Fundamentals	91.44	5	2
38	Kendra	Bonnicksen	Composition - Intermediate	88.91	6	2
37	George	Chavez	Composition - Fundamentals	88.54	7	2
37	Marianne	Wier	Composition - Fundamentals	87.4	8	2
37	David	Hamilton	Composition - Fundamentals	86.33	9	3
37	Steve	Pundt	Composition - Fundamentals	82.58	10	3
38	Doris	Hartwig	Composition - Intermediate	81.66	11	3
37	Michael	Viescas	Composition - Fundamentals	77.59	12	3
38	Elizabeth	Hallmark	Composition - Intermediate	72.88	13	4
37	Karen	Smith	Composition - Fundamentals	72.05	14	4

（续）

Subject ID	Stud FirstName	Stud LastName	Subject Name	Grade	Rank	Quintile
37	Betsy	Stadick	Composition - Fundamentals	71.09	15	4
37	Kerry	Patterson	Composition - Fundamentals	70	16	5
38	Sarah	Thompson	Composition - Intermediate	67.6	17	5
38	Richard	Lum	Composition - Intermediate	67.19	18	5

❖ **说明**　敏锐的读者可能注意到了，这个查询返回的结果与第 20 章的查询 CH20_English_Student_Quintiles 返回的结果稍有不同。

函数 NTILE()将行划分为指定数量的分组，如果行数不是分组数的整数倍，前面的分组将比后面的分组大。例如，如果有 10 个数据行，NTILE(2)将把这些行分成 2 个大小相同的分组，但 NTILE(3)将在第一个分组中包含 4 行，在第二个和第三个分组中分别包含 3 行。这里就出现了这种情况：需要将 18 行分成 5 个分组，因此这些分组分别包含 4、4、4、3 和 3 行。这就是这个示例与 CH20_English_Student_Quintiles 使用的五分算法的不同之处。

这两种方法没有优劣之分，如果有 20 名学生，它们计算得到的五分区间将相同。

22.5　结合使用窗口和聚合函数

本章前面说过，你可将第 12 章介绍的聚合函数与 OVER()子句结合起来使用，完整的语法图如图 22-2 所示。前面演示了如何将函数 COUNT(*)和 SUM()同 OVER()子句结合起来使用，这里再简单地介绍一下如何将它们同谓词 ROWS 和 RANGE 结合起来使用。

别忘了，谓词 ROWS 和 RANGE 限定了聚合函数将使用哪些数据。结合使用谓词 ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW 和函数 COUNT(*)并没有太大的意义：这让你能够计算当前行前面有多少行；换而言之，它提供的值与前面介绍过的函数 ROW_NUMBER()相同。结合使用谓词 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW 和函数 COUNT(*)可能更有用些：可提供基于范围的移动总计。

“指出每个订单有多少个订单详情行。我要看到订单号、订购的商品和商品数量以及订单详情行移动总计。”

转换/整理　Select ~~the~~ OrderNumber, ProductName, COUNT(*) OVER partition by OrderNumber, COUNT(*) OVER order by OrderNumber rows between unbounded preceding and current row, COUNT(*) OVER order by OrderNumber ranges between unbounded preceding and current row from ~~the~~ Orders ~~table~~ inner join~~ed with the~~ Order_Details ~~table~~ on Order_Details.OrderNumber = Orders.OrderNumber inner join~~ed with the~~ Products ~~table~~ on Products.ProductNumber = OrderDetails.ProductNumber（依次基于 Order_Details.OrderNumber = Orders.OrderNumber 将 Orders 表内连接到 Order_Details 表、基于 Products.ProductNumber = OrderDetails.ProductNumber 内连接到 Products 表，生成一个结果集；选择 OrderNumber、ProductName 和三个通过计算得到的数字。这三个数字分别是这样计算得到的：将结果集按 OrderNumber 分区，并计算当前分区包含的行数；将结果集按 OrderNumber 排序，并计算从开头到当前位置有多少行；将结果集按 OrderNumber 排序，并计算 OrderNumber 小于等于当前行的行数）

SQL

```
SELECT O.OrderNumber AS OrderNo, P.ProductName,
  COUNT(*) OVER (
    PARTITION BY O.OrderNumber
  ) AS Total,
  COUNT(*) OVER (
    ORDER BY O.OrderNumber
    ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT
      ROW
  ) AS TotalUsingRows,
  COUNT(*) OVER (
    ORDER BY O.OrderNumber
```

```
      RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT
        ROW
    ) AS TotalUsingRange
  FROM Orders AS O
    INNER JOIN Order_Details AS OD
      ON OD.OrderNumber = O.OrderNumber
    INNER JOIN Products AS P
      ON P.ProductNumber = OD.ProductNumber;
```

这个查询返回的结果类似于下表（这个查询保存在示例数据库 Sales Orders 中，名为 CH22_Order_Counts_ByInvoice_ROWS_RANGE）。

OrderNo	ProductName	Total	Total UsingRows	Total UsingRange
1	Trek 9000 Mountain Bike	7	1	7
1	Viscount Mountain Bike	7	2	7
1	GT RTS-2 Mountain Bike	7	3	7
1	ProFormance ATB All-Terrain Pedal	7	4	7
1	Dog Ear Aero-Flow Floor Pump	7	5	7
1	Glide-O-Matic Cycling Helmet	7	6	7
1	Ultimate Export 2G Car Rack	7	7	7
2	X-Pro All Weather Tires	2	8	9
2	Ultimate Export 2G Car Rack	2	9	9
3	Trek 9000 Mountain Bike	8	10	17
3	Viscount Mountain Bike	8	11	17
		<< 其他行 >>		

如你所见，1 号订单包含 7 种商品，因此在这些商品对应的行中，Total 列都为 7。同理，2 号订单包含两种商品，因此相关行的 Total 列都为 2（这里只显示了 3 号订单的一部分，但可以看到，与之相关的两行的 Total 列值相同）。

前面说过，TotalUsingRows 列包含的只是一系列相连的数字，与使用函数 ROW_NUMBER()提供的值相同。

最后，TotalUsingRange 列显示了基于订单的商品数移动总计。换而言之，在与 1 号订单相关的全部 7 行中，该列的值都为 7；接下来，在与 2 号订单相关的两行中，该列的值都为 9（与 1 号订单相关的 7 行加上与 2 号订单相关的两行）。从 Total 列的值可知，有 8 行与 3 号订单相关，因此这些行的 TotalUsingRange 列值为 17（7 + 2 + 8）。

当然，其他聚合函数也很有用。例如，你可能想知道每位顾客所下订单的详情，包括订单总价。

"列出所有的订单，包括下单顾客的姓名、订单号、商品名、订购数量、报价和订单总价。"

转换/整理

Select ~~the~~ CustFirstName || ' ' ||, CustLastName, OrderNumber, ProductName, QuantityOrdered, QuotedPrice, ~~and~~ SUM(QuotedPrice) OVER partition by OrderNumber from ~~the~~ Orders ~~table~~ inner join~~ed with the~~ Order_Details ~~table~~ on Order_Details.OrderNumber = Orders.OrderNumber inner join~~ed with the~~ Customers ~~table~~ on Customers.CustomerID = Orders.CustomerID inner join~~ed with the~~ Products ~~table~~ on Products.ProductNumber = OrderDetails.ProductNumber（依次基于 Order_Details.OrderNumber = Orders.OrderNumber 将 Orders 表内连接到 Order_Details 表、基于 Customers.CustomerID = Orders.CustomerID 内连接到 Customers 表、基于 Products.ProductNumber = OrderDetails.ProductNumber 内连接到 Products 表；根据 OrderNumber 分区，并从每个分区中选择 CustFirstName || ' ' || CustLastName、OrderNumber、ProductName、QuantityOrdered、QuotedPrice 和 SUM(QuotedPrice)）

SQL

```
SELECT C.CustFirstName || ' ' || C.CustLastName
  AS Customer,
  O.OrderNumber AS Order, P.ProductName,
  OD.QuantityOrdered AS Quantity,
  OD.QuotedPrice AS Price,
  SUM(OD.QuotedPrice) OVER (
    PARTITION BY O.OrderNumber
  ) AS OrderTotal
FROM Orders AS O
  INNER JOIN Order_Details AS OD
```

```
    ON OD.OrderNumber = O.OrderNumber
  INNER JOIN Customers AS C
    ON C.CustomerID = O.CustomerID
  INNER JOIN Products AS P
    ON P.ProductNumber = OD.ProductNumber
```

这个查询的结果类似于下表（这个查询保存在示例数据库 Sales Orders 中，名为 CH22_Order_Totals_ByInvoice）。

Customer	Order	ProductName	Quantity	Price	OrderTotal
David Smith	1	Trek 9000 Mountain Bike	2	1200.00	3863.85
David Smith	1	Viscount Mountain Bike	3	635.00	3863.85
David Smith	1	GT RTS-2 Mountain Bike	4	1650.00	3863.85
David Smith	1	ProFormance ATB All-Terrain Pedal	1	28.00	3863.85
David Smith	1	Dog Ear Aero-Flow Floor Pump	3	55.00	3863.85
David Smith	1	Glide-O-Matic Cycling Helmet	5	121.25	3863.85
David Smith	1	Ultimate Export 2G Car Rack	6	174.60	3863.85
Suzanne Viescas	2	X-Pro All Weather Tires	4	24.00	204.00
Suzanne Viescas	2	Ultimate Export 2G Car Rack	4	180.00	204.00
William Thompson	3	Trek 9000 Mountain Bike	5	1164.00	3824.29
William Thompson	3	Viscount Mountain Bike	5	615.95	3824.29
		<< 其他行 >>			

从中可知，有 7 行与 1 号订单相关，它们的 OrderTotal 列值都相同；有两行与 2 号订单相关，它们的 OrderTotal 列值也都相同（这里只显示了 3 号订单的一部分，但显示的两行的 OrderTotal 列值相同）。

我们再来看一个使用谓词 ROWS 的示例（虽然我看不出这个示例有什么实际用途，但希望它能够帮你更深入地理解谓词 ROWS 的工作原理）。

> "列出所有的演出合约，包括顾客、开始日期、合约价格、相邻 3 行（当前行及其前后各一行）的合约价格总和。另外，显示按顾客分区时相邻 3 行（当前行及其前后各一行）的合约价格总和。"

转换/整理　Select ~~the~~ CustFirstName || ' ' ||, CustLastName, StartDate, ContractPrice, SUM(ContractPrice) OVER order by CustLastName, CustFirstName rows between 1 preceding and 1 following ~~and~~ SUM(ContractPrice) OVER partition by CustLastName, CustFirstName order by CustLastName, CustFirstName rows between 1 preceding and 1 following from ~~the~~ Engagements ~~table~~ inner join~~ed with the~~ Customers ~~table~~ on Customers.CustomerID = Engagements.CustomerID［基于 Customers.CustomerID = Engagements.CustomerID 将 Engagements 表内连接到 Customers 表，并选择 CustFirstName || ' ' || CustLastName、StartDate、ContractPrice 和两个合约价格总和。这两个合约价格总和分别是这样计算得到的：根据 CustLastName 和 CustFirstName 进行排序，再计算相邻 3 行（当前行及其前后一行）的合约价格总和；按 CustLastName 和 CustFirstName 分区并排序，再计算相邻 3 行（当前行及其前后一行）的合约价格总和］

SQL

```
SELECT C.CustFirstName || ' ' || C.CustLastName AS Customer,
  E.StartDate, E.ContractPrice,
  SUM(E.ContractPrice) OVER (
    ORDER BY C.CustLastName, C.CustFirstName
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
  ) AS SumOf3,
  SUM(E.ContractPrice) OVER (
    PARTITION BY C.CustLastName, C.CustFirstName
    ORDER BY C.CustLastName, C.CustFirstName
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
  ) AS PartitionedSumOf3
FROM Engagements AS E
  INNER JOIN Customers AS C
    ON C.CustomerID = E.CustomerID;
```

这个查询的结果类似于下表（这个查询保存在示例数据库 Entertainment Agency 中，名为 CH22_Odd_Contract_Sums）。

Customer	StartDate	ContractPrice	SumOf3	PartitionedSumOf3
Matt Berg	2017-09-02	200.00	1330.00	1330.00
Matt Berg	2017-09-12	1130.00	3180.00	3180.00
Matt Berg	2017-09-19	1850.00	5655.00	5655.00
Matt Berg	2017-10-14	2675.00	6450.00	6450.00
Matt Berg	2017-10-23	1925.00	6150.00	6150.00
Matt Berg	2017-12-31	1550.00	3795.00	3795.00
Matt Berg	2018-01-09	320.00	3540.00	3540.00
Matt Berg	2018-02-12	1670.00	3840.00	3840.00
Matt Berg	2018-02-24	1850.00	4320.00	3520.00
Peter Brehm	2018-02-17	800.00	2940.00	1090.00
Peter Brehm	2018-01-07	290.00	1860.00	1860.00
Peter Brehm	2018-01-30	770.00	1380.00	1380.00
Peter Brehm	2018-02-27	320.00	1590.00	1590.00
Peter Brehm	2017-12-10	500.00	4620.00	4620.00
Peter Brehm	2017-10-07	3800.00	5070.00	5070.00
Peter Brehm	2017-09-18	770.00	6120.00	4570.00
Zachary Ehrlich	2017-10-03	1550.00	3690.00	2920.00
Zachary Ehrlich	2017-09-19	1370.00	4170.00	4170.00
Zachary Ehrlich	2017-10-08	1250.00	3030.00	3030.00
		<< 其他行 >>		

SumOf3 列包含的是当前行、前一行和后一行的 ContractPrice 总和。在第 1 行中，这列的值为 1330.00。它没有前一行，它的 ContractPrice 列值为 200.00，它后面一行的 ContractPrice 列值为 1130.00，因此总和为 1330.00。对于第 2 行，前一行的 ContractPrice 列值为 200.00，当前行的 ContractPrice 列值为 1130.00，后一行的 ContractPrice 列值为 1850.00，因此 SumOf3 列值为 3180.00（200.00 + 1130.00 + 1850.00）。接着往下看第 10 行（Peter Brehm 的第 1 行），它有前一行（虽然这行是另一位顾客的，但别忘了我只指定了 ORDER BY，而没有指定 PARTITION BY），其 ContractPrice 列值为 1850.00，而当前行和后一行的 ContractPrice 列值分别为 800.00 和 290.00，因此该行的 SumOf3 列值为 2940.00（1850.00 + 800.00 + 290.00）。

前 8 行都是同一位顾客（Matt Berg）的，因此它们各自的 SumOf3 和 PartitionedSumOf3 列值相同，但第 9 行是 Matt Berg 的最后一行，因此在分区的情况下没有后一行，这意味着 PartitionedSumOf3 列值为前一行的 ContractPrice 列值（1670.00）加上当前行的 ContractPrice 列值（1850.00），即 3520.00。接下来是第 10 行（Peter Brehm 的第 1 行），在分区的情况下，它没有前一行，因此 PartitionedSumOf3 列值为当前行的 ContractPrice 列值（800.00）加上下一行的 ContractPrice 列值（290.00），即 1090.00。

22.6 语句举例

现在你知道了如何使用窗口函数编写查询，还知道了使用窗口函数可解决的一些问题类型，下面来看一些示例，它们都使用了一个或多个窗口函数。这些示例都来自示例数据库，演示了如何使用窗口函数。

❖ **说明** 本书前言提醒过，在你使用的数据库系统中，行的排列顺序不一定与本书示例中显示的相同——除非使用 ORDER BY 子句。即便指定了排序方式，在数据库系统返回的结果中，对于 ORDER BY 子句中未包含的列，根据它们的排列顺序也可能不同，这是因为优化方法因数据库系统而异。

如果你在 Microsoft SQL Server 中运行示例，则从视图中选择行时，将不会遵循其中的 ORDER BY 子句指定的排列顺序。要遵循 ORDER BY 子句指定的排列顺序，必须打开视图的设计，并在显示界面中执行它。

另外，当你使用 GROUP BY 子句时，数据库系统返回的结果集看起来像是根据该子句中指定的列进行了排序。这是因为有些优化器首先会在内部对数据排序，以提高 GROUP BY 的处理速度。别忘了，如果要按特定的顺序排列，必须使用 ORDER BY 子句。

在 SQL 语法行后面，还列出了这些操作返回的示例结果集。在结果集前面显示了名称，这是本书配套网站提供的示例数据库中给查询指定的名称。每个查询都存储在相应的示例数据库中，名称以 CH22 打头。可按本书前言中的说明在你的计算机中加载这些示例，并尝试执行它们。

❖ **说明**　别忘了，这些示例中使用的所有列名和表名都来自示例数据库。有关这些数据库的结构，请参阅附录 B。这些示例很多都使用了复杂的连接，你使用的数据库系统处理这些查询的方式可能不同。因此，我列出的前几行可能与你获得的结果不完全相同，但总行数应该相同。为简化流程，我在下面的示例中将转换和整理合二为一了。

22.6.1　ROW_NUMBER 使用示例

- **Entertainment Agency 数据库**

"给我一个演出合约清单，列出每个演出合约的开始日期、顾客姓名和演唱组合，依次给所有演出合约编号，并依次给开始日期相同的演出合约编号。"

转换/整理

Select ~~the~~ StartDate, CustFirstName || ' ' || CustLastName AS Customer, EntStartName AS Entertaine~~, the~~ ROW_NUMBER() order~~ed~~ by StartDate AS Number ~~and the~~ ROW_NUMBER() partition~~ed~~ by StartDate ~~and~~ order~~ed~~ by StartDate AS NumberByDate from ~~the~~ Engagements ~~table~~ inner join~~ed with the~~ Entertainers ~~table~~ on Entertainers.EntertainerID = Engagements.EntertainerID inner join~~ed with the~~ Customers ~~table~~ on Customers.CustomerID = Engagements.CustomerID［基于 Entertainers.EntertainerID = Engagements.EntertainerID 将 Engagements 表内连接到 Entertainers 表、基于 Customers.CustomerID = Engagements.CustomerID 内连接到 Customers 表；选择 StartDate、CustFirstName || ' ' || CustLastName（作为 Customer）、EntStageName（作为 Entertainer）、Number 和 NumberByDate。Number 和 NumberByDate 分别是这样计算得到的：按 StartDate 排序，再使用 ROW_NUMBER()获取行号；按 StartDate 分区并排序，再使用 ROW_NUMBER()获取行号］

SQL

```
SELECT ROW_NUMBER() OVER (
    ORDER BY Engagements.StartDate
  ) AS Number,
  Engagements.StartDate,
  ROW_NUMBER() OVER (
    PARTITION BY Engagements.StartDate
    ORDER BY Engagements.StartDate
  ) AS NumberByDate,
Customers.CustFirstName || ' ' ||
   Customers.CustLastName AS Customer,
  Entertainers.EntStageName AS Entertainer
FROM Engagements
  INNER JOIN Entertainers
    ON Entertainers.EntertainerID =
      Engagements.EntertainerID
  INNER JOIN Customers
    ON Customers.CustomerID =
      Engagements.CustomerID;
```

CH22_Engagements_Numbered（111 行）

Number	StartDate	Number ByDate	Customer	Entertainer
1	2017-09-02	1	Matt Berg	Jim Glynn
2	2017-09-11	1	Doris Hartwig	Jazz Persuasion
3	2017-09-11	2	Mark Rosales	Country Feeling
4	2017-09-12	1	Dean McCrae	Topazz
5	2017-09-12	2	Liz Keyser	Jim Glynn

（续）

Number	StartDate	Number ByDate	Customer	Entertainer
6	2017-09-12	3	Matt Berg	JV & the Deep Six
7	2017-09-16	1	Elizabeth Hallmark	Country Feeling
8	2017-09-18	1	Elizabeth Hallmark	JV & the Deep Six
9	2017-09-18	2	Peter Brehm	Modern Dance
10	2017-09-19	1	Matt Berg	Coldwater Cattle Company
11	2017-09-19	2	Zachary Ehrlich	Saturday Revue
12	2017-09-19	3	Mark Rosales	Carol Peacock Trio
13	2017-09-25	1	Doris Hartwig	Country Feeling
14	2017-09-25	2	Liz Keyser	Caroline Coie Cuartet
15	2017-09-30	1	Deb Waldal	Saturday Revue
16	2017-09-30	2	Sarah Thompson	Jim Glynn
		<< 其他行 >>		
110	2018-02-27	1	Peter Brehm	Julia Schnebly
111	2018-03-04	1	Mark Rosales	JV & the Deep Six

- **Recipes 数据库**

“给我一个带编号的菜品清单，编号包含两种：总体编号和菜品类型内编号。将菜品按菜品类型和菜品名排序。请不要遗漏不包含任何菜品的菜品类型。”

转换/整理

Select ~~the~~ RecipeClassDescription, ~~the~~ RecipeTitle, ~~the~~ ROW_NUMBER() ordered by RecipeClassDescription, RecipeTitle AS OverallNumber ~~and the~~ ROW_NUMBER() partition~~ed~~ by RecipeClassDescription ~~and~~ order~~ed~~ by RecipeTitle AS ClassNumber from ~~the~~ Recipe_Classes ~~table~~ left join~~ed with the~~ Recipes ~~table~~ on Recipes.RecipeClass ID = Recipe_Classes.RecipeClassID（基于 Recipes.RecipeClass ID = Recipe_Classes.RecipeClassID 将 Recipe_Classes 表左外连接到 Recipes 表；选择 RecipeClassDescription、RecipeTitle、OverallNumber 和 ClassNumber。OverallNumber 和 ClassNumber 分别是这样计算得到的：按 RecipeClassDescription 和 RecipeTitle 排序，再使用 ROW_NUMBER()获取行号；按 RecipeClassDescription 分区并按 RecipeTitle 排序，再使用 ROW_NUMBER()获取行号）

SQL

```
SELECT ROW_NUMBER() OVER (
    ORDER BY RC.RecipeClassDescription,
            R.RecipeTitle
  ) AS OverallNumber,
  RC.RecipeClassDescription,
  ROW_NUMBER() OVER (
    PARTITION BY RC.RecipeClassDescription
    ORDER BY R.RecipeTitle
  ) AS NumberInClass,
  R.RecipeTitle
FROM Recipe_Classes AS RC
  LEFT JOIN Recipes AS R
    ON R.RecipeClassID = RC.RecipeClassID;
```

CH22_Recipe_Classes_Numbered（16 行）

Overall Number	RecipeClass Description	Number InClass	RecipeTitle
1	Dessert	1	Coupe Colonel
2	Dessert	2	Trifle
3	Hors d’oeuvres	1	Machos Nachos
4	Hors d’oeuvres	2	Salsa Buena
5	Main course	1	Fettuccini Alfredo
6	Main course	2	Huachinango Veracruzana(Red Snapper, Veracruz style)

（续）

Overall Number	RecipeClass Description	Number InClass	RecipeTitle
7	Main course	3	Irish Stew
8	Main course	4	Pollo Picoso
9	Main course	5	Roast Beef
10	Main course	6	Salmon Filets in Parchment Paper
11	Main course	7	Tourtière (French-Canadian Pork Pie)
12	Salad	1	Mike's Summer Salad
13	Soup	1	NULL
14	Starch	1	Yorkshire Pudding
15	Vegetable	1	Asparagus
16	Vegetable	2	Garlic Green Beans

22.6.2　RANK、DENSE_RANK 和 PERCENT_RANK 使用示例

- **Sales Orders 数据库**

“根据相关联的订单数对所有员工排名。”

转换/整理

Select ~~the~~ EmployeeID, EmpFirstName || ' ' || EmpLastName AS EmployeeName, COUNT(DISTINCT OrderNumber) AS OrdersReceived and RANK() OVER order~~ed~~ by COUNT(DISTINCT OrderNumber) DESC from ~~the~~ Employees ~~table~~ inner join~~ed with the~~ Orders ~~table~~ ON Orders.EmployeeID = Employees.EmployeeID inner join~~ed with the~~ Order_Details ~~table~~ on OrderDetails.OrderNumber = Orders.OrderNumber, group~~ed~~ by EmployeeID, EmpFirstName ~~and~~ EmpLastName［依次基于 Orders.EmployeeID = Employees.EmployeeID 将 Employees 表内连接到 Orders 表、基于 OrderDetails.OrderNumber = Orders.OrderNumber 内连接到 Order_Details 表，并根据 EmployeeID、EmpFirstName 和 EmpLastName 进行分组；从每个分组中选择 EmployeeID、EmpFirstName || ' ' || EmpLastName（作为 EmployeeName）、COUNT(DISTINCT OrderNumber)（作为 OrdersReceived）和排名，其中排名是这样计算得到的：根据 COUNT(DISTINCT OrderNumber)降序排列分组，再使用 RANK()获取分组的排名］

SQL

```
SE LECT E.EmployeeID, E.EmpFirstName || ' ' ||
  E.EmpLastName AS Employee,
  COUNT(DISTINCT O.OrderNumber) AS OrdersReceived,
  RANK() OVER (
    ORDER BY COUNT(DISTINCT O.OrderNumber) DESC
  ) AS Rank
FROM Employees AS E
  INNER JOIN Orders AS O
    ON O.EmployeeID = E.EmployeeID
  INNER JOIN Order_Details AS OD
    ON OD.OrderNumber = O.OrderNumber
GROUP BY E.EmployeeID, E.EmpFirstName,
   E.EmpLastName;
```

CH22_Employee_Sales_Ranked（8 行）

EmployeeID	Employee	OrdersReceived	Rank
707	Kathryn Patterson	138	1
708	Susan McLain	129	2
702	Mary Thompson	117	3
704	Carol Viescas	117	3
705	Kirk DeGrasse	115	5
701	Ann Patterson	109	6
706	David Viescas	104	7
703	Matt Berg	104	7

你可能会问：为何在上面的 SQL 中使用 Order_Details 表呢？原因是我知道 Orders 表中的有些行在 Order_Details

表中没有与之匹配的行，而我要将没有任何商品的订单排除在外，因此我将 Orders 表内连接到 Order_Details 表，从而排除这些空订单。

- **School Scheduling 数据库**

“根据到 2018 年 1 月 1 日为学校服务的年限对教员排名。我不希望排名是跳跃的。”

（别忘了，CH19_Length_Of_Service 演示了如何计算服务年数。）

转换/整理 Select ~~the~~ StaffID, StfFirstName || ' ' || StfLastName AS StaffName, LengthOfService, ~~and~~ DENSE_RANK() OVER order~~ed~~ by LengthOfService DESC from ~~table~~ Staff［从 Staff 表中选择 StaffID、StfFirstName || ' ' || StfLastName（作为 StaffName）、LengthOfService 和排名，其中排名是这样计算得到的：根据 LengthOfService 降序排列，再使用 DENSE_RANK()获取排名］

SQL

```
SELECT StaffID, StfFirstName || ' ' || StfLastname
  AS StaffName,
  YEAR(CAST('2018-01-01' As Date)) -
    YEAR(DateHired) -
    (CASE WHEN Month(DateHired) < 10
       THEN 0
    WHEN Month(DateHired) > 10
       THEN 1
    WHEN Day(DateHired) > 1
       THEN 1
    ELSE 0 END
 ) AS LengthOfService,
DENSE_RANK() OVER (ORDER BY YEAR
  (CAST('2018-01-01' As Date)) - YEAR(DateHired) -
  (CASE WHEN Month(DateHired) < 10
     THEN 0
    WHEN Month(DateHired) > 10
      THEN 1
    WHEN Day(DateHired) > 1
      THEN 1
    ELSE 0 END
   ) DESC) AS Rank
FROM Staff;
```

CH22_Staff_Service_Ranked（27 行）

StaffID	StaffName	LengthOfService	Rank
98036	Sam Abolrous	27	1
98062	Caroline Coie	27	1
98025	Carol Viescas	26	2
98028	Alaina Hallmark	26	2
98010	Jeffrey Smith	26	2
98011	Ann Patterson	26	2
98007	Gary Hallmark	25	3
98020	Jim Glynn	25	3
98043	Kathryn Patterson	25	3
98052	Katherine Ehrlich	25	3
98013	Deb Waldal	24	4
98014	Peter Brehm	24	4
98005	Suzanne Viescas	24	4
98048	Joyce Bonnicksen	24	4
98059	Maria Patterson	24	4
98040	Jim Wilson	23	5
98030	Liz Keyser	22	6
98063	Kirk DeGrasse	22	6
98064	Luke Patterson	21	7
98055	Alastair Black	21	7
98057	Joe Rosales III	21	7

（续）

StaffID	StaffName	LengthOfService	Rank
98012	Robert Brown	21	7
98021	Tim Smith	21	7
98019	Mariya Sergienko	20	8
98045	Michael Hernandez	20	8
98053	Caleb Viescas	18	9
98042	David Smith	18	9

22.6.3 NTILE 使用示例

- **Bowling League 数据库**

“根据队员的平均让分得分按从高到低的顺序对球队排名，并将球队划分到四分区间中。”

转换/整理

Select ~~the~~ TeamName, AVG(HandiCapScore) ~~and~~ NTILE(4) OVER order~~ed~~ by AVG(HandiCapScore) DESC from ~~the~~ Teams ~~table~~ inner join~~ed with the~~ Bowlers ~~table~~ ON Bowlers.TeamID = Teams.TeamID inner join~~ed with the~~ Bowler_Scores ~~table~~ on BowlerScores.BowlerID = Bowlers.BowerID, group~~ed~~ by TeamName（依次基于 Bowlers.TeamID = Teams.TeamID 将 Teams 表内连接到 Bowlers 表、基于 BowlerScores.BowlerID = Bowlers.BowerID 内连接到 Bowler_Scores 表，并根据 TeamName 分组；从每个分组中选择 TeamName、AVG(HandiCapScore)和四分区间，其中四分区间是这样计算得到的：根据 AVG(HandiCapScore)降序排列分组，再使用 NTILE(4)获取四分区间）

SQL

```
SELECT Teams.TeamName,
  ROUND(AVG(Bowler_Scores.HandiCapScore), 0)
          AS AvgTeamHandicap,
  NTILE(4) OVER (
    ORDER BY ROUND(AVG(Bowler_Scores.HandiCap-
             Score), 0)
    DESC) AS Quartile
FROM Teams INNER JOIN Bowlers
    ON Bowlers.TeamID = Teams.TeamID
  INNER JOIN Bowler_Scores
    ON Bowler_Scores.BowlerID = Bowlers.BowlerID
GROUP BY Teams.TeamName;
```

CH22_Teams_In_Quartiles（8 行）

TeamName	AvgTeamHandicap	Quartile
Marlins	196	1
Barracudas	196	1
Manatees	196	2
Swordfish	195	2
Orcas	195	3
Dolphins	195	3
Sharks	195	4
Terrapins	195	4

你肯定会问：结果怎么会这样呢？由于球队之间的平均让分得分几乎相同，数据库系统先根据平均让分得分将球队排序，再将前两行的排名指定为 1，接下来两行指定为 2，再接下来两行指定为 3，最后两行指定为 4。实际上，在值相等的情况下，排序结果因数据库系统而异。如果你在 Microsoft SQL Server 和 PostgreSQL 中分别运行这个示例，将发现它们选择的前两名截然不同。

- **Entertainment Agency 数据库**

“根据签约数量对演唱组合排名，并将它们划分到三分区间中。请不要遗漏没有签订任何演出合约的演唱组合。”

转换/整理

Select ~~the~~ EntStageName, COUNT(Engagements.EntertainerID), ~~and~~ NTILE(3) OVER order~~ed~~ by COUNT(Engagements.EntertainerID) DESC from ~~the~~ Entertainers ~~table~~ left join~~ed with the~~ Engagements ~~table~~ ON Engagements. EntertainerID = Entertainers.EntertainerID, grouped by EntStageName（基于 Engagements. EntertainerID = Entertainers.EntertainerID 将 Entertainers 表左外连接到 Engagements 表，并根据 EntStageName 分组；从每个分组中，选择 EntStageName、COUNT(Engagements.EntertainerID) 和三分区间，其中三分区间是这样计算得到的：根据 COUNT(Engagements.EntertainerID)将分组排序，再使用 NTILE(3) 获取三分区间）

SQL

```
SELECT Entertainers.EntStageName AS Entertainer,
  COUNT(Engagements.EntertainerID) AS Gigs,
  NTILE(3) OVER (
    ORDER BY COUNT(Engagements.EntertainerID) DESC
  ) AS [Group]
FROM Entertainers
  LEFT JOIN Engagements
    ON Engagements.EntertainerID = Entertainers.
       EntertainerID
GROUP BY Entertainers.EntStageName;
```

CH22_Entertainer_3_Groups（13 行）

Entertainer	Gigs	Group
Country Feeling	15	1
Carol Peacock Trio	11	1
Caroline Coie Cuartet	11	1
JV & the Deep Six	10	1
Modern Dance	10	1
Saturday Revue	9	2
Jim Glynn	9	2
Julia Schnebly	8	2
Coldwater Cattle Company	8	2
Jazz Persuasion	7	3
Topazz	7	3
Susan McLain	6	3
Katherine Ehrlich	0	3

22.6.4 聚合函数使用示例

- **Bowling League 数据库**

“对于每个球队，显示其队长参加的所有比赛局次的详细信息，包括所属联赛的日期和地点、让分得分、是否获胜。另外，指出每个球队的队长赢了多少局及其平均让分得分。”

转换/整理

Select ~~the~~ TeamName, BowlerFirstName || ' ' || BowlerLastName AS Captain, TourneyDate, TourneyLocation, HandiCapScore, CASE WonGame WHEN 1 THEN ‘Won’ ELSE ‘Lost’ END As WonGame, SUM(CAST(WonGame AS INT) OVER partition~~ed~~ by TeamName AS TotalWindws, AVG(HandiCapScore) OVER(partitioned by TeamName) AS AvgHandicap from ~~table~~ Teams inner join~~ed with the~~ Bowlers ~~table~~ ON Bowler.BowlerID = Teams.CaptainID inner join~~ed with the~~ Bowler_Scores ~~table~~ on Bowler_Scores.BowlerID = Teams. CaptainID inner join~~ed with the~~ Match_Games ~~table~~ on MatchGames.MatchID = Bowler_Scores.MatchID and MatchGames.GameNumber = Bowler_Scores.GameNumber inner join~~ed with the~~ Tourney_Matches ~~table~~ on Tourney_Matches. MatchID = Match_Games.MatchID inner join~~ed with the~~ Tournaments ~~table~~ on Tournaments.TourneyID = Tourney_Matches.TourneyID［依次基于 Bowler.BowlerID = Teams.CaptainID 将 Teams 表内连接到 Bowlers 表、基于 Bowler_Scores.BowlerID = Teams.CaptainID 内连接到 Bowler_Scores 表、基于 MatchGames.MatchID = Bowler_Scores.MatchID 和 MatchGames.GameNumber = Bowler_Scores.GameNumber 内连接到 Match_Games 表、基于 Tourney_Matches. MatchID = Match_Games.MatchID 内连接到 Tourney_Matches 表、基于 Tournaments.TourneyID = Tourney_ Matches.TourneyID 内连接到 Tournaments 表；选择 TeamName、BowlerFirstName || ' ' || BowlerLastName（作为 Captain）、TourneyDate、TourneyLocation、HandiCapScore、WonGame、TotalWindws 和 AvgHandicap，其中 WonGame、TotalWindws 和 AvgHandicap 分别是这样计算得到的：如果 WonGame 为 1 就返回 Won，否则返回 Lost；根据 TeamName 分区，将分区中每行的 WonGame 的转换为整数，再计算它们的总和；根据 TeamName 分区，再计算分区中 HandiCapScore 的平均值］

SQL

```
SELECT Teams.TeamName,
  B.BowlerFirstName || ' ' || B.BowlerLastName
    AS Captain,
  T.TourneyDate, T.TourneyLocation,
  BS.HandiCapScore,
  CASE BS.WonGame
    WHEN 1 THEN 'Won'
    ELSE 'Lost'
  END AS WonGame,
  SUM(CAST(BS.WonGame AS INT)) OVER (
    PARTITION BY Teams.TeamName
  ) AS TotalWins,
  AVG(BS.HandiCapScore) OVER (
    PARTITION BY Teams.TeamName
  ) AS AvgHandicap
FROM Teams
  INNER JOIN Bowlers AS B
    ON B.BowlerID = Teams.CaptainID
  INNER JOIN Bowler_Scores AS BS
    ON BS.BowlerID = B.BowlerID
  INNER JOIN Match_Games AS MG
    ON MG.MatchID = BS.MatchID
    AND MG.GameNumber = BS.GameNumber
  INNER JOIN Tourney_Matches AS TM
    ON TM.MatchID = MG.MatchID
  INNER JOIN Tournaments AS T
    ON T.TourneyID = TM.TourneyID;
```

CH22_Comparing_Team_Captains（420 行）

TeamName	Captain	Tourney Date	Tourney Location	HandiCap Score	Won Game	Total Wins	Avg Handicap
Barracudas	Richard Sheskey	2017-09-04	Red Rooster Lanes	195	Won	25	194
Barracudas	Richard Sheskey	2017-09-04	Red Rooster Lanes	191	Lost	25	194
Barracudas	Richard Sheskey	2017-09-04	Red Rooster Lanes	195	Won	25	194
Barracudas	Richard Sheskey	2017-09-11	Thunderbird Lanes	201	Won	25	194
Barracudas	Richard Sheskey	2017-09-11	Thunderbird Lanes	201	Won	25	194
Barracudas	Richard Sheskey	2017-09-11	Thunderbird Lanes	188	Won	25	194
Barracudas	Richard Sheskey	2017-09-18	Bolero Lanes	196	Won	25	194
Barracudas	Richard Sheskey	2017-09-18	Bolero Lanes	189	Lost	25	194
Barracudas	Richard Sheskey	2017-09-18	Bolero Lanes	193	Won	25	194
Barracudas	Richard Sheskey	2017-09-25	Imperial Lanes	190	Lost	25	194
Barracudas	Richard Sheskey	2017-09-25	Imperial Lanes	188	Lost	25	194
Barracudas	Richard Sheskey	2017-09-25	Imperial Lanes	198	Won	25	194
Barracudas	Richard Sheskey	2017-10-02	Sports World Lanes	195	Lost	25	194
Barracudas	Richard Sheskey	2017-10-02	Sports World Lanes	189	Won	25	194
Barracudas	Richard Sheskey	2017-10-02	Sports World Lanes	190	Lost	25	194
Barracudas	Richard Sheskey	2017-10-09	Totem Lanes	191	Lost	25	194
Barracudas	Richard Sheskey	2017-10-09	Totem Lanes	200	Won	25	194
Barracudas	Richard Sheskey	2017-10-09	Totem Lanes	191	Lost	25	194
Barracudas	Richard Sheskey	2017-10-16	Acapulco Lanes	190	Won	25	194
Barracudas	Richard Sheskey	2017-10-16	Acapulco Lanes	200	Won	25	194
Barracudas	Richard Sheskey	2017-10-16	Acapulco Lanes	187	Lost	25	194
<< 其他行 >>							

❖ **说明**　原本也可根据 BowlerFirstName 和 BowlerLastName（而不是 TeamName）分区，但我觉得通常最好不要根据人名分区，因为它们并不总是独一无二的。也可根据 BowlerID 或 CaptainID 进行分区。

● **Sales Orders 数据库**

必须承认，这个 ROWS 使用示例也有点矫揉造作，提供它只是为了让你理解 ROWS 的工作原理。

"对于每个订单，列出下单顾客、订购的商品及其数量，以及订购的商品总数。另外，对于订单中的每种商品，将其与前一种商品和后一种商品编为一组，并显示这些商品的总订购数量、最少订购数量和最大订购数量。"

转换/整理

Select ~~the~~ CustFirstName || ' ' || CustLastName, OrderNumber, ProductName, QuantityOrdered, SUM(QuantityOrdered) OVER(partition~~ed~~ by OrderNumber) AS TotalQuantity, SUM(QuantityOrdered) OVER(partition~~ed~~ by OrderNumber order~~ed~~ by OrderNumber ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS Quantity3, MIN(QuantityOrdered) OVER (partition~~ed~~ by OrderNumber order~~ed~~ by OrderNumber ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS Minimum3, MAX(QuantityOrdered) OVER(partition~~ed~~ by OrderNumber order~~ed~~ by OrderNumber ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS Maximum3 from ~~the~~ Orders ~~table~~ inner join~~ed with the~~ Order_Details ~~table~~ ON Order_Details.OrderNumber = Orders. OrderNumber inner join~~ed with the~~ Customers ~~table~~ on Customers.CustomerID = Orders.CustomerID inner join~~ed with the~~ Products ~~table~~ on Products.ProductNumber = Order_Details.ProductNumber（依次基于 Order_Details.OrderNumber = Orders.OrderNumber 将 Orders 表内连接到 Order_Details 表、基于 Customers.CustomerID = Orders.CustomerID 内连接到 Customers 表、基于 Products.ProductNumber = Order_Details.ProductNumber 内连接到 Products 表。选择 CustFirstName || ' ' || CustLastName、OrderNumber、ProductName、QuantityOrdered、TotalQuantity、Quantity3、Minimum3 和 Maximum3，其中 TotalQuantity、Quantity3、Minimum3 和 Maximum3 分别是这样计算得到的：根据 OrderNumber 进行分区，并计算分区的 QuantityOrdered 总和；根据 OrderNumber 进行分区，对于分区中的每一行，计算它自己、前一行和后一行的 QuantityOrdered；根据 OrderNumber 进行分区，对于分区中的每一行，找出它自己、前一行和后一行中最小的 QuantityOrdered 值；根据 OrderNumber 进行分区，对于分区中的每一行，找出它自己、前一行和后一行中最大的 QuantityOrdered 值）

SQL

```
SELECT O.OrderNumber,
  C.CustFirstName || ' ' || C.CustLastName AS
Customer,
  O.OrderNumber,
  P.ProductName,
  OD.QuantityOrdered,
  SUM(OD.QuantityOrdered) OVER (
    PARTITION BY O.OrderNumber
  ) AS TotalQuantity,
  SUM(OD.QuantityOrdered) OVER (
    PARTITION BY O.OrderNumber
    ORDER BY O.OrderNumber
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
  ) AS Quantity3,
  MIN(OD.QuantityOrdered) OVER (
    PARTITION BY O.OrderNumber
    ORDER BY O.OrderNumber
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
  ) AS Minimum3,
  MAX(OD.QuantityOrdered) OVER (
    PARTITION BY O.OrderNumber
    ORDER BY O.OrderNumber
    ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
  ) AS Maximum3
FROM Orders AS O
  INNER JOIN Order_Details AS OD
    ON OD.OrderNumber = O.OrderNumber
  INNER JOIN Customers AS C
    ON C.CustomerID = O.CustomerID
  INNER JOIN Products AS P
    ON P.ProductNumber = OD.ProductNumber;
```

CH22_Orders_Min_Max（3973 行）

Order Number	Customer	Product Name	Quantity Ordered	Total Quantity	Quantity3	Minimum3	Maximum3
1	David Smith	Trek 9000 Mountain Bike	2	24	5	2	3
1	David Smith	Viscount Mountain Bike	3	24	9	2	4
1	David Smith	GT RTS-2 Mountain Bike	4	24	8	1	4
1	David Smith	ProFormance ATB All- Terrain Pedal	1	24	8	1	4
1	David Smith	Dog Ear Aero-Flow Floor Pump	3	24	9	1	5
1	David Smith	Glide-O-Matic Cycling Helmet	5	24	14	3	6

（续）

Order Number	Customer	Product Name	Quantity Ordered	Total Quantity	Quantity3	Minimum3	Maximum3
1	David Smith	Ultimate Export 2G Car Rack	6	24	11	5	6
2	Suzanne Viescas	X-Pro All Weather Tires	4	8	8	4	4
2	Suzanne Viescas	Ultimate Export 2G Car Rack	4	8	8	4	4
3	William Thompson	Trek 9000 Mountain Bike	5	28	10	5	5
3	William Thompson	Viscount Mountain Bike	5	28	11	1	5
3	William Thompson	GT RTS-2 Mountain Bike	1	28	8	1	5
3	William Thompson	ProFormance ATB All- Terrain Pedal	2	28	6	1	3
3	William Thompson	Dog Ear Aero-Flow Floor Pump	3	28	8	2	3
3	William Thompson	Glide-O-Matic Cycling Helmet	3	28	11	3	5
3	William Thompson	True Grip Competition Gloves	5	28	12	3	5
3	William Thompson	Cosmic Elite Road Warrior Wheels	4	28	9	4	5
4	Andrew Cencini	Trek 9000 Mountain Bike	4	19	7	3	4
			<< 其他行 >>				

QuantityOrdered 列可能很好理解。第一个订单（编号为 1，顾客为 David Smith）包含 7 种商品（总共 24 件）：Trek 9000 Mountain Bikes（2 件）、Viscount Mountain Bikes（3 件）、GT RTS-2 Mountain Bikes（4 件）、ProFormance ATB All-Terrain Pedal（1 件）、Dog Ear AeroFlow Floor Pumps（3 件）、Glide-O-Matic Cycling Helmets（5 件）和 Ultimate Export 2G Car Racks（6 件）。第二个订单（编号为 2，顾客为 Suzanne Viescas）包含 4 件 X-Pro All Weather Tires 和 4 件 Ultimate Export 2G Car Racks，总共 8 件商品。这些列都是非常标准的。

其他 3 列就不那么标准了。对于每个订单中的商品，我要求将相邻的三种分成一组，并计算每组的总订购数量、最少订购数量和最大订购数量。还记得 ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING 的工作原理吗？由于我按订单进行了分区，因此第一个订单的第一种商品（Trek 9000 Mountain Bike）没有前一行，所有只考虑当前行（订购数量为 2）和下一行（订购数量为 3）。这些订购数量的总和为 5，最小订购数量为 2，最大订购数量为 3。下一行（Viscount Mountain Bike）有前一行，因此需要考虑 3 个订购数量： 2（前一行）、3（当前行）和 4（下一行）。这些订购数量的总和为 9，最小值为 2，最大值为 3。我们直接跳到第 7 行。同样，由于我按订单进行了分区，因此这一行没有下一行，所以只需考虑前一行（订购数量为 5）和当前行（订购数量为 6）。这两个订购数量的总和为 11，最小值为 5，最大值为 6。

由于第二个订单只有两行，因此对于这两行，都只需考虑两行（它们的订购数量都为 4）。这些订购数量的总和为 8，最小值和最大值都为 4。

对于第三个订单，其中的 Trek 9000 Mountain Bike 行没有前一行，因此对于这一行，只需考虑它自己和下一行（Viscount Mountain Bike）。同理，Cosmic Elite Road Warrior Wheels 行没有下一行，因此对于这一行，只需考虑它自己和前一行（True Grip Competition Gloves）。

- **School Scheduling 数据库**

“对于每个科目，列出最高成绩；同时，列出每个科目类别和所有科目类别的最高成绩。”

转换/整理	Select ~~the~~ CategoryID, SubjectCode, SubjectName, MAX(Grade) OVER(partion~~ed~~ by SubjectID) AS SubjectMax, MAX(Grade) OVER(partition~~ed~~ by CategoryID) AS CategoryMax, MAX(Grade) OVER() AS OverallMax from ~~the~~ Subjects ~~table~~ inner join~~ed with the~~ Classes ~~table~~ ON Classes.ClassID = Subjects.ClassID inner join~~ed with the~~ Student_Schedules ~~table~~ on Student_Schedules.ClassID = Classes. ClassID where ClassStatus = 2（依次基于 Classes.ClassID = Subjects.ClassID 将 Subjects 表内连接到 Classes 表、基于 Student_Schedules.ClassID = Classes.ClassID 内连接到 Student_Schedules 表，并选择 ClassStatus = 2 的行。选择不同的 CategoryID、SubjectCode、SubjectName 以及 SubjectMax、CategoryMax 和 OverallMax，其中 SubjectMax、CategoryMax 和 OverallMax 分别是这样计算得到的：根据 SubjectID 进行分区，并在每个分区中找出最大的 Grade；根据 CategoryID 进行分区，并在每个分区中找出最大的 Grade；找出整个结果集中最大的 Grade）

```
SQL         SELECT DISTINCT Subjects.CategoryID,
              Subjects.SubjectCode, Subjects.SubjectName,
              MAX(Student_Schedules.Grade) OVER (
                PARTITION BY Subjects.SubjectID
              ) AS SubjectMax,
              MAX(Student_Schedules.Grade) OVER (
                PARTITION BY Subjects.CategoryID
              ) AS CategoryMax,
              MAX(Student_Schedules.Grade) OVER (
              ) AS OverallMax
            FROM Subjects
              INNER JOIN Classes
                ON Classes.SubjectID = Subjects.SubjectID
              INNER JOIN Student_Schedules
                ON Student_Schedules.ClassID = Classes.ClassID
            WHERE Student_Schedules.ClassStatus = 2;
```

CH22_Top_Marks（14 行）

CategoryID	SubjectCode	SubjectName	Subject Max	Category Max	Overall Max
ACC	ACC 210	Financial Accounting Fundamentals I	91.12	91.12	99.83
ART	ART 100	Introduction to Art	99.83	99.83	99.83
ART	ART 210	Computer Art	87.65	99.83	99.83
ART	ART 251	Art History	97.81	99.83	99.83
BIO	BIO 101	General Biology	94.54	94.54	99.83
CHE	CHE 139	Fundamentals of	98.31	98.31	99.83
CIS	CIS 101	Chemistry	98.01	98.01	99.83
ENG	ENG 101	Microcomputer	98.07	98.07	99.83
ENG	ENG 102	Applications	88.91	98.07	99.83
HIS	HIS 111	Composition -	87.14	87.14	99.83
MAT	MAT 080	Fundamentals	93.19	94.33	99.83
MAT	MAT 097	Composition -	94.33	94.33	99.83
MUS	MUS 100	Intermediate	97.84	97.84	99.83
MUS	MUS 101	U.S. History to 1877	86.57	97.84	99.83

22.7 小结

本章首先阐述了要以不同于本书前面介绍的方式聚合数据的原因，并列举了一个示例。

接下来探索了将数据划分成窗口的方式，指出了 OVER()子句是打开窗口函数大门的钥匙，并阐述了 GROUP BY 和 OVER()的差别所在。

接着介绍了 PARTITION BY、ORDER BY、ROWS（RANGE）对结果的影响。

- 谓词 PARTITION BY 将查询返回的结果集分成数据子集。
- 谓词 ORDER BY 决定了将按什么顺序评估各行。
- 谓词 ROWS（RANGE）只能与聚合函数一起使用，决定了将使用分区中哪些行的数据来执行计算。

然后指出窗口函数包含本书前面介绍的所有聚合函数，还有其他一些函数：ROW_NUMBER、RANK、DENSE_RANK、PERCENT_RANK 和 NTILE。之后介绍了如何将谓词 PARTITION BY 和 ORDER BY 与 OVER()子句结合起来使用。

最后列举了多个使用窗口函数解决现实问题的示例。

下一节列出了多个你可独自解决的问题。

22.8 练习

- **Bowling League 数据库**

  (1) 根据最高的队员原始得分将球队划归到五分区间。

解决方案见 CH22_Team_Quartiles_Best_RawScore（8 行）。

(2) 列出联盟中所有的投球手，根据姓和名排序，并给每个投球手一个整体编号和一个球队内编号。
解决方案见 CH22_Bowler_Numbers（32 行）。

(3) 根据平均让分得分对联盟的所有投球手排名，显示标准排名以及非跳跃排名。
解决方案见 CH22_Bowler_Ranks（32 行）。

- **Entertainment Agency 数据库**

(1) 根据签订的演出合约的总价对所有经纪人排名，千万不要遗漏没有签订任何演出合约的经纪人。
解决方案见 CH22_Agent_Ranks（9 行）。

(2) 列出每个演唱组合的全部演出合约，包含演唱组合的艺名、顾客姓名、开始日期以及每个演唱组合签订的演出合约总数。
解决方案见 CH22_Entertainer_Engagements（111 行）。

(3) 列出所有的演唱组合及其成员，并在演唱组合内给每个成员编号。
解决方案见 CH22_Entertainer_Lists（40 行）。

- **Recipes 数据库**

(1) 列出所有的菜品；对于每个菜品，列出其使用的所有食材以及总共使用了多少种食材。
解决方案见 CH22_Recipe_Ingredient_Counts（88 行）。

(2) 列出所有的食材；对于每种食材，列出所有使用了它的菜品以及有多少种菜品使用了它。
解决方案见 CH22_Ingredient_Recipe_Counts（88 行）。

(3) 生成一个带编号的食材清单。对于每种食材，给它指定一个整体编号以及食材类型内编号。在食材类型中，按字母顺序排列食材。请不要遗漏不包含任何食材的食材类型。
解决方案见 CH22_Ingredients_By_Ingredient_Class（83 行，包括 4 个不包含任何食材的食材类型）。

- **Sales Orders 数据库**

(1) 显示每个订单的总价，并按总价从高到低的顺序对订单排名。
解决方案见 CH22_Order_Totals_RankedByInvoiceTotal（933 行）。

(2) 生成一个清单，其中包含每种商品类型及售出的该类商品的总价，并在一列中显示售出的所有商品的总价。
解决方案见 CH22_Sales_Totals（6 行）。
如果你知道有些类型的商品一件也没卖出，你将以怎样不同的方式编写这个查询？（在示例数据库中，确实没有一件也没卖出的商品类型，但你可新增一种没有任何销售记录的商品类型，看看你编写的查询是否管用。）这种情况下的解决方案见 CH22_Sales_Totals_Handle_Nulls。如果你没有新增商品类型，它返回的结果将与 CH22_Sales_Totals 一样（6 行）；如果你新增了一种商品类型，它将返回 7 行。

(3) 根据下单数量对顾客排名，别遗漏没有下过单的顾客。
解决方案见 CH22_Customer_Order_Counts_Ranked（28 行，包括没有下过单的顾客 Jeffrey Tirekicker）。

- **School Scheduling 数据库**

(1) 根据修完的课程数量对学生排名。
解决方案见 CH22_Student_Class_Totals_Rank。（18 行，差别是不是很小？）

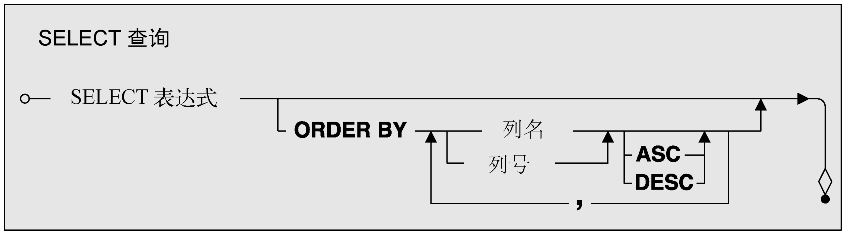
(2) 根据讲授的课程数量对教员排名。
解决方案见 CH22_Staff_Class_Totals_Rank（22 行）。

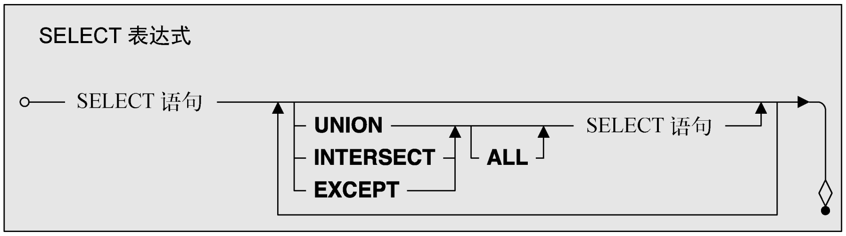
(3) 根据修完的课程的平均成绩将学生分成 3 组。
解决方案见 CH22_Student_AverageGrade_Groups（18 行）。

(4) 对于每位学生，列出其修完的所有课程及成绩，并显示其修完的所有课程的平均成绩。另外，对于每门课程，将它自己、前一门课程和下一门课程作为一组，并找出该组的平均成绩、最高成绩和最低成绩。
解决方案见 CH22_Marks_Min_Max（68 行）。

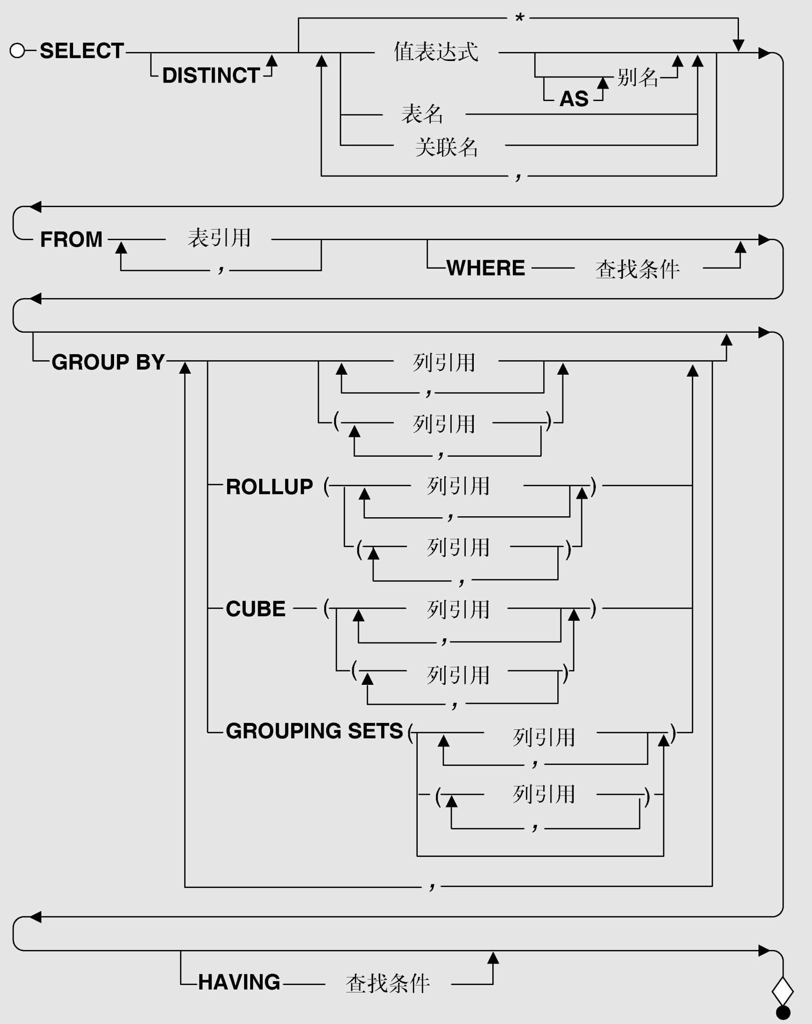
附录 A SQL 标准语法图

这里列出了本书涉及的所有 SQL 语法的完整语法图。

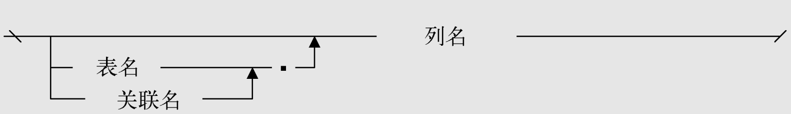




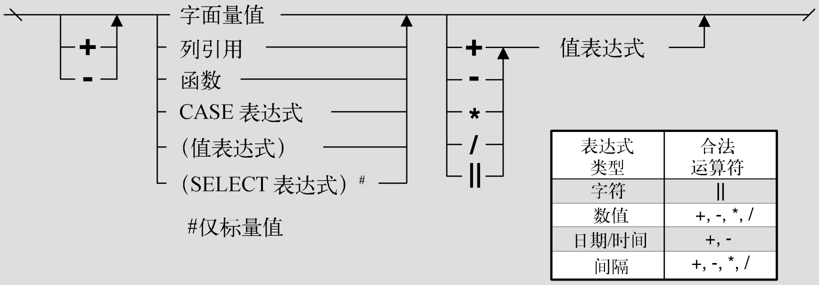
SELECT 语句



列引用



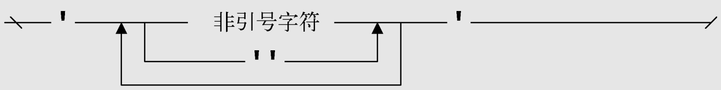
值表达式



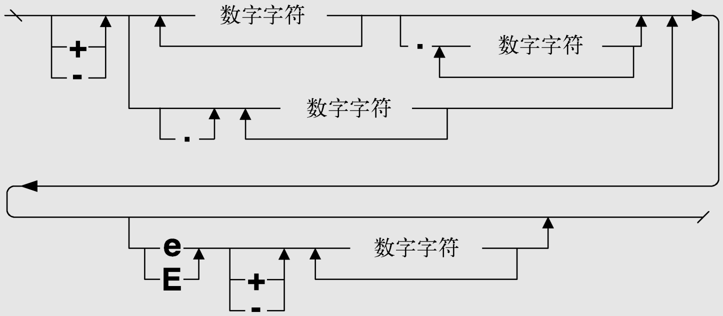
表达式类型	合法运算符
字符	||
数值	+, -, *, /
日期/时间	+, -
间隔	+, -, *, /

字面量值

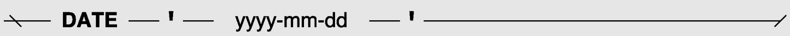
字符串



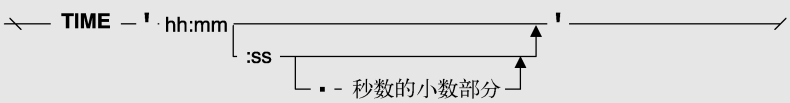
数值



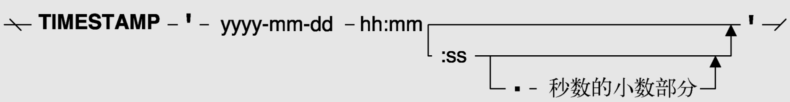
日期

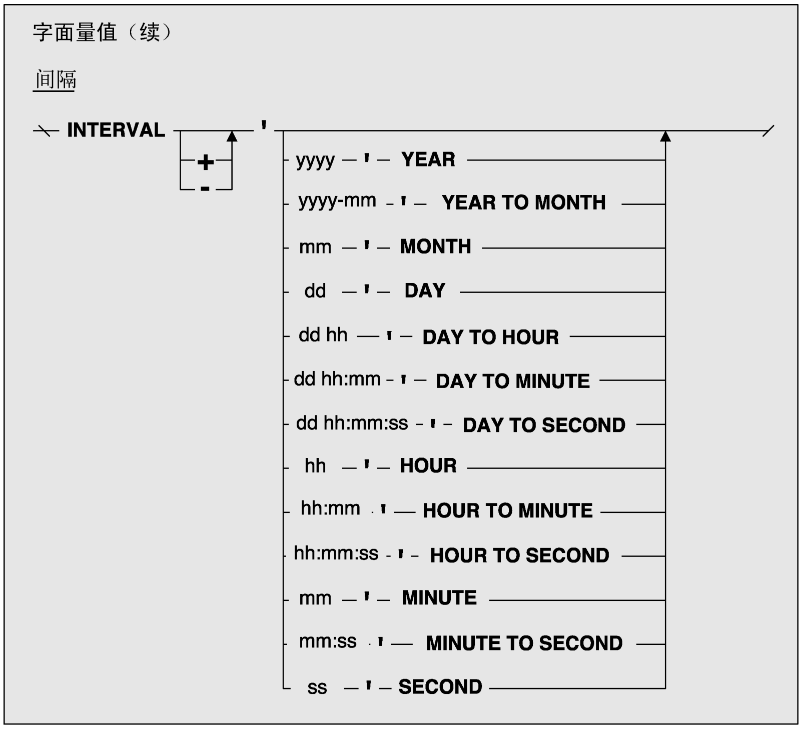


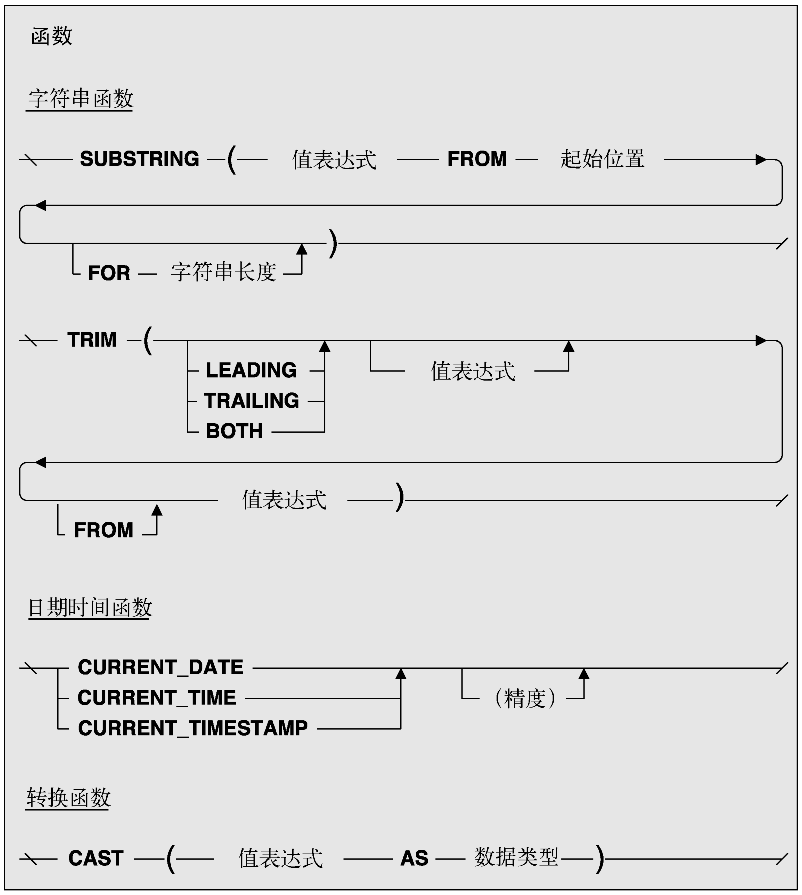
时间



时间戳

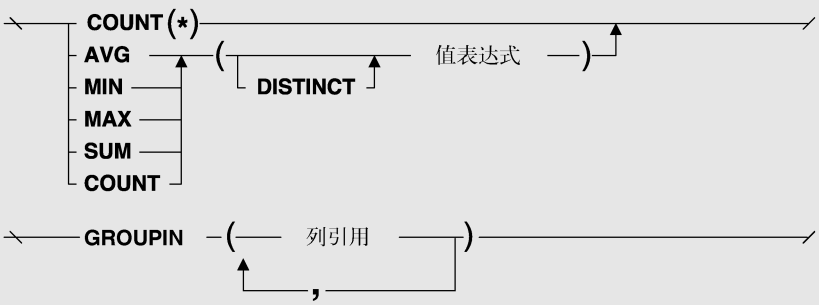



字面量值（续）
间隔
INTERVAL
+
-
'
yyyy ' YEAR
yyyy-mm ' YEAR TO MONTH
mm ' MONTH
dd ' DAY
dd hh ' DAY TO HOUR
dd hh:mm ' DAY TO MINUTE
dd hh:mm:ss ' DAY TO SECOND
hh ' HOUR
hh:mm ' HOUR TO MINUTE
hh:mm:ss ' HOUR TO SECOND
mm ' MINUTE
mm:ss ' MINUTE TO SECOND
ss ' SECOND

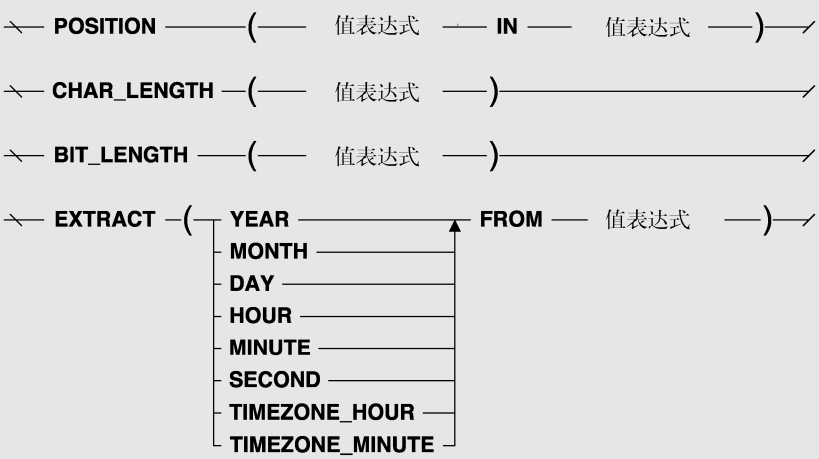

函数
字符串函数
SUBSTRING (值表达式 FROM 起始位置
FOR 字符串长度)
TRIM (
LEADING
TRAILING
BOTH
值表达式
FROM
值表达式)
日期时间函数
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
（精度）
转换函数
CAST (值表达式 AS 数据类型)

函数（续）

集合函数（聚合函数）

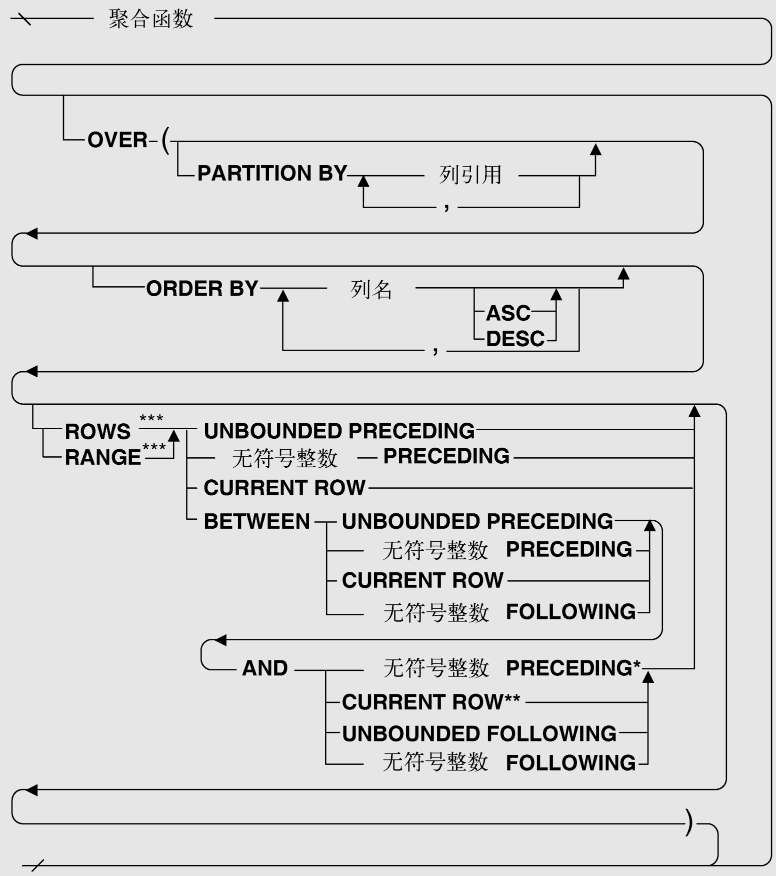


数值函数



函数（续）

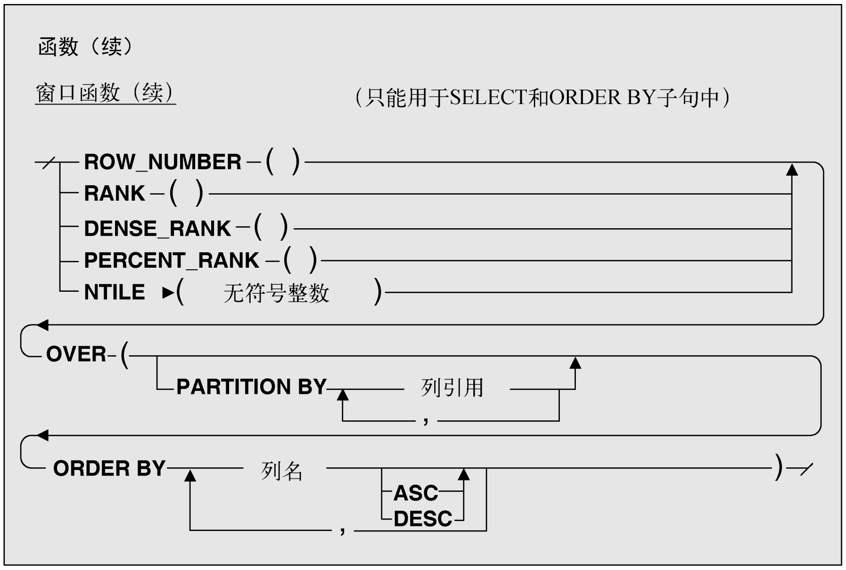
窗口函数　　　　（只能用于SELECT和ORDER BY子句中）

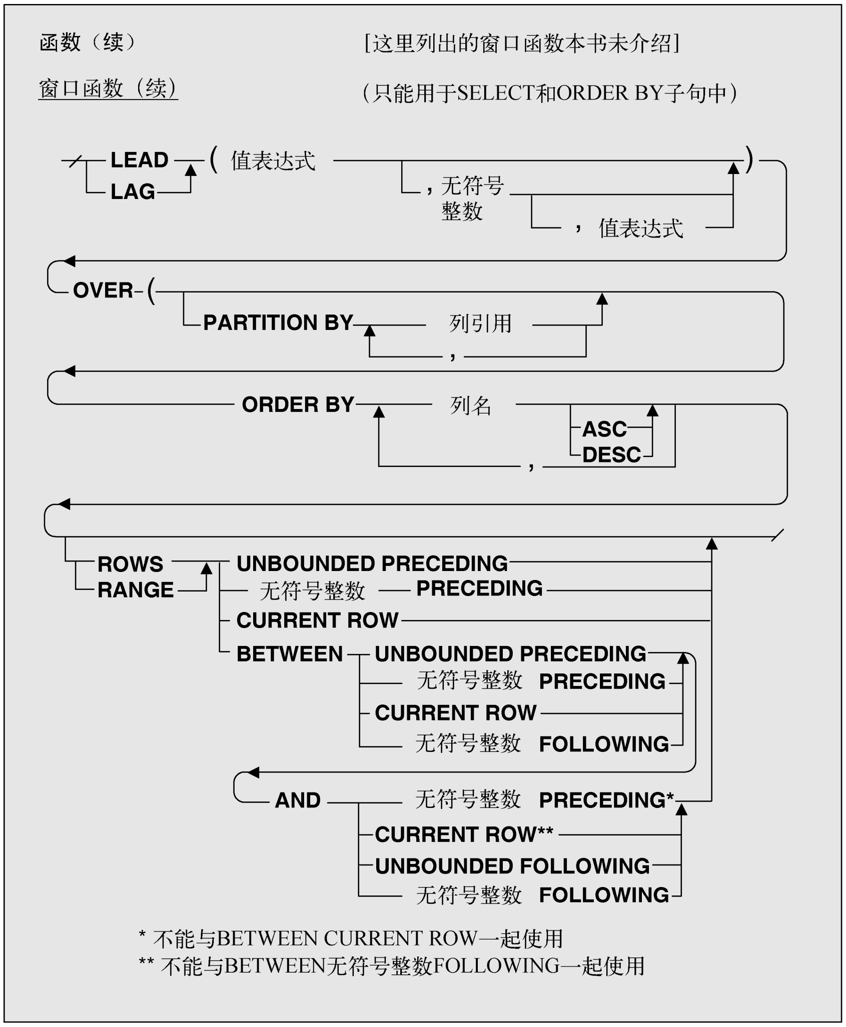


* 不能与BETWEEN CURRENT ROW一起使用

** 不能与BETWEEN 无符号整数FOLLOWING一起使用

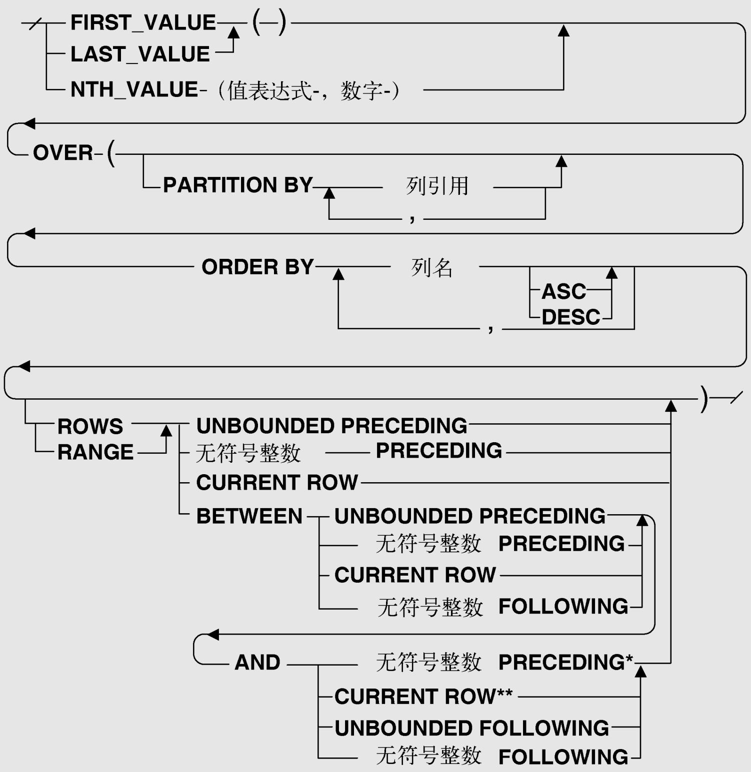
*** 必须包含ORDER BY子句


函数（续）
窗口函数（续）
（只能用于SELECT和ORDER BY子句中）
ROW_NUMBER ( )
RANK ( )
DENSE_RANK ( )
PERCENT_RANK ( )
NTILE (
无符号整数
)
OVER (
PARTITION BY
列引用
,
ORDER BY
列名
ASC
DESC
,
)


函数（续）
[这里列出的窗口函数本书未介绍]
窗口函数（续）
（只能用于SELECT和ORDER BY子句中）
LEAD
LAG
(值表达式
, 无符号整数
, 值表达式
)
OVER (
PARTITION BY
列引用
,
ORDER BY
列名
ASC
DESC
,
ROWS
RANGE
UNBOUNDED PRECEDING
无符号整数
PRECEDING
CURRENT ROW
BETWEEN
UNBOUNDED PRECEDING
无符号整数
PRECEDING
CURRENT ROW
无符号整数
FOLLOWING
AND
无符号整数
PRECEDING*
CURRENT ROW**
UNBOUNDED FOLLOWING
无符号整数
FOLLOWING
* 不能与BETWEEN CURRENT ROW一起使用
** 不能与BETWEEN无符号整数FOLLOWING一起使用

函数（续）　　　　[这里列出的窗口函数本书未介绍]

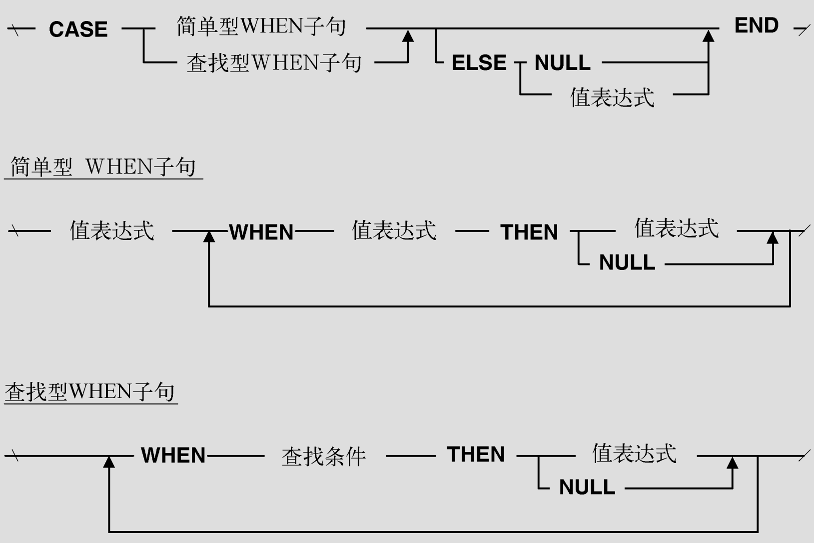
窗口函数（续）　　　　（只能用于SELECT和ORDER BY子句中）



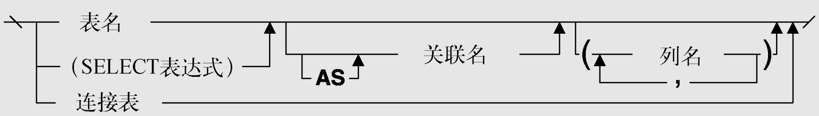
* 不能与BETWEEN CURRENT ROW一起使用

** 不能与BETWEEN无符号整数FOLLOWING一起使用

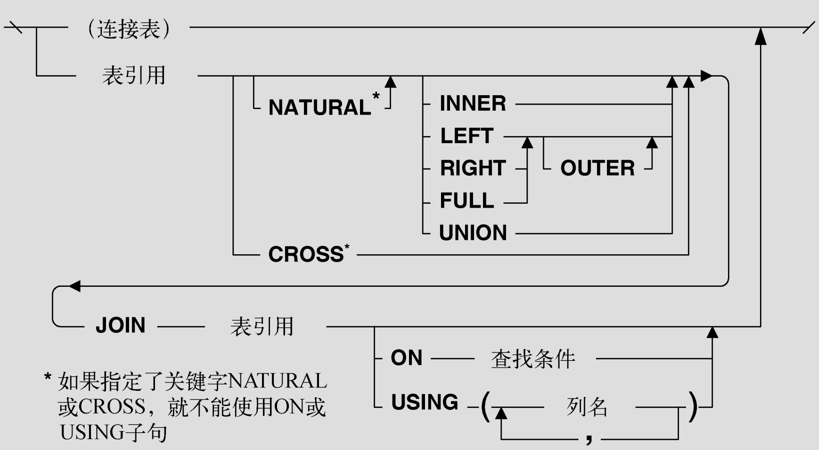
CASE 表达式

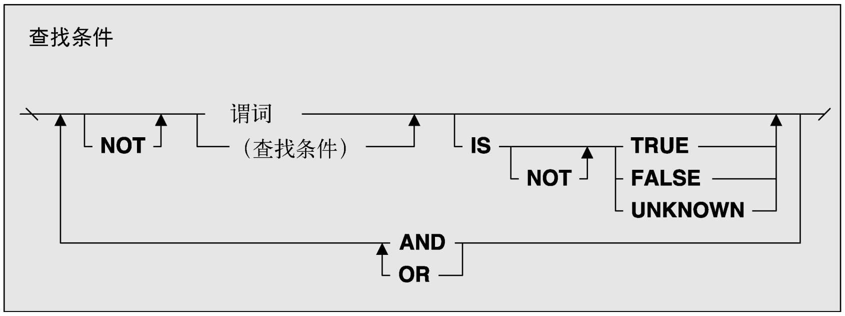


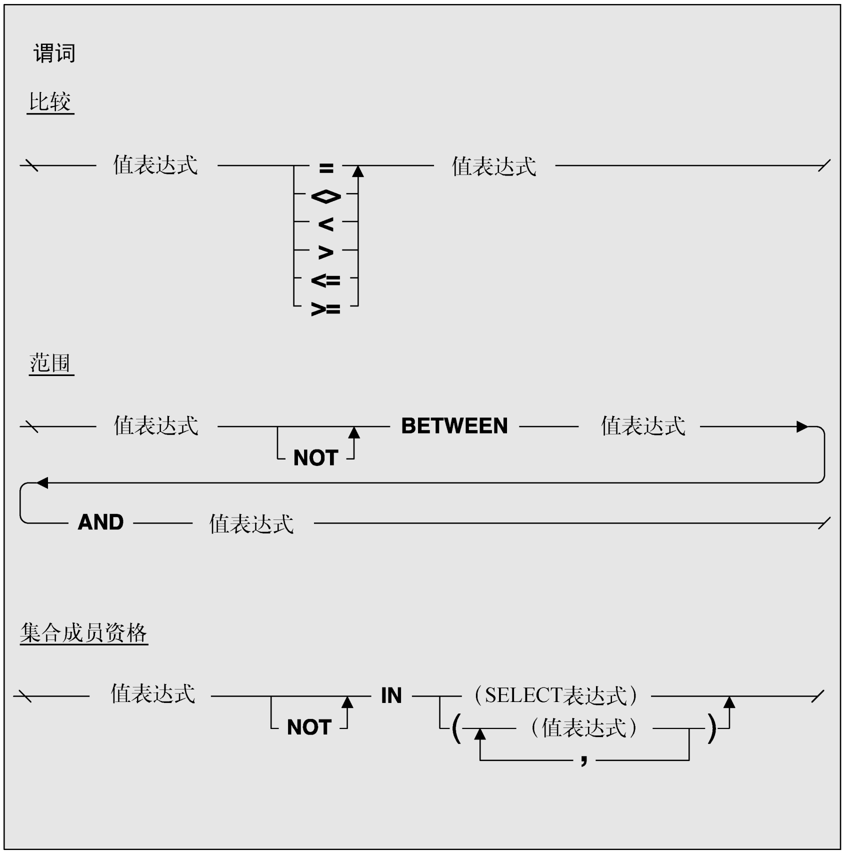
表引用

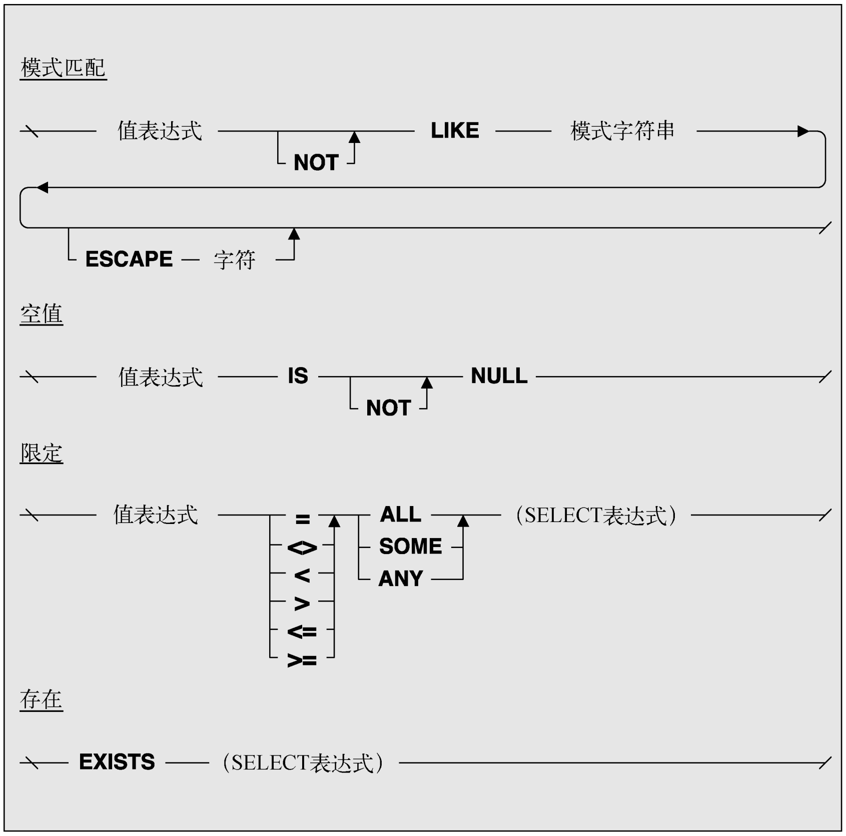


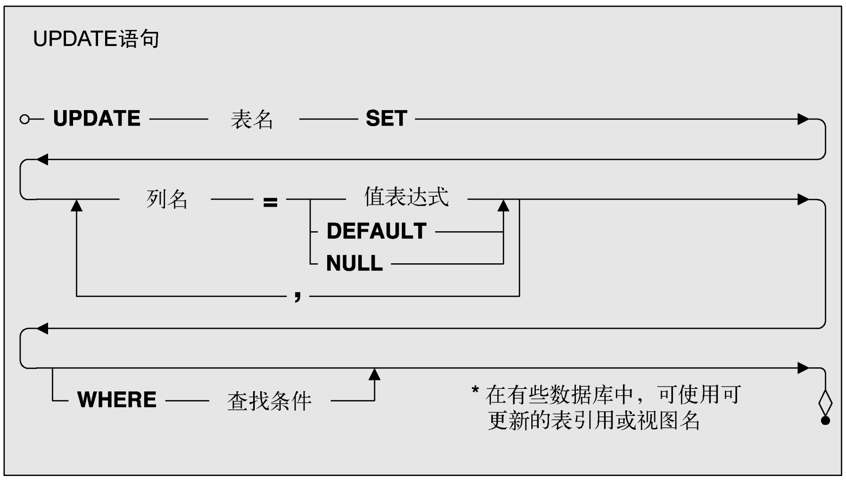
连接表

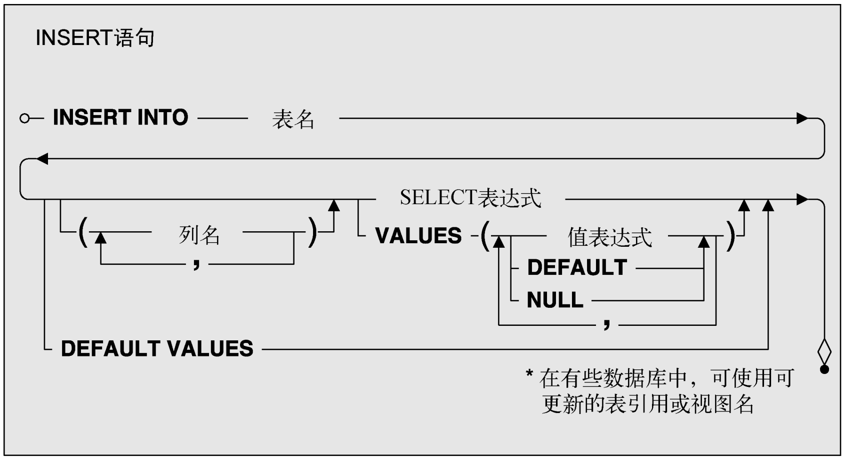



查找条件
NOT
谓词
（查找条件）
IS
NOT
TRUE
FALSE
UNKNOWN
AND
OR


谓词
比较
值表达式
=
<>
<
>
<=
>=
值表达式
范围
值表达式
NOT
BETWEEN
值表达式
AND
值表达式
集合成员资格
值表达式
NOT
IN
（SELECT表达式）
(
（值表达式）
,
)


模式匹配
值表达式
NOT
LIKE
模式字符串
ESCAPE
字符
空值
值表达式
IS
NOT
NULL
限定
值表达式
=
<>
<
>
<=
>=
ALL
SOME
ANY
（SELECT表达式）
存在
EXISTS
（SELECT表达式）


UPDATE语句
UPDATE
表名
SET
列名
=
值表达式
DEFAULT
NULL
,
WHERE
查找条件
* 在有些数据库中，可使用可
更新的表引用或视图名

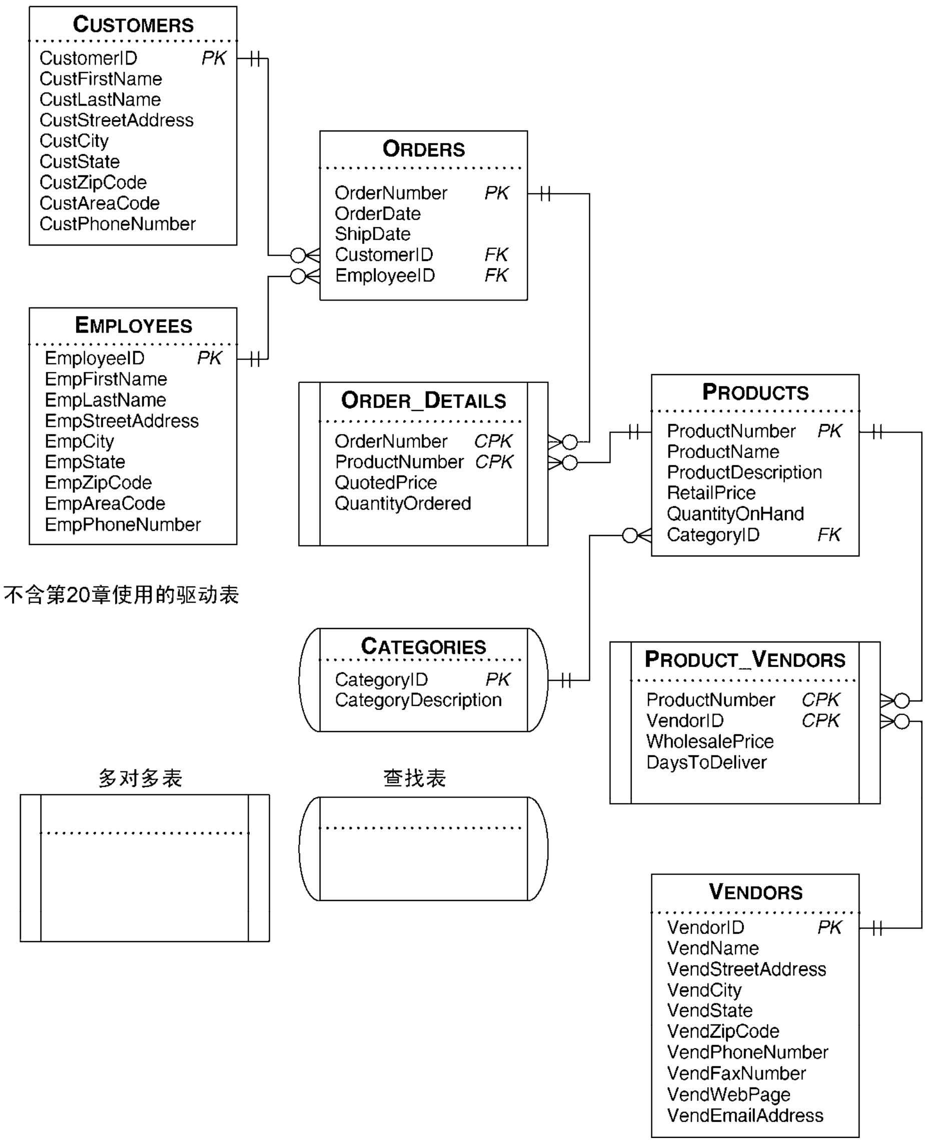

INSERT语句
INSERT INTO
表名
SELECT表达式
(
列名
)
,
VALUES
(
值表达式
DEFAULT
NULL
)
,
DEFAULT VALUES
* 在有些数据库中，可使用可更新的表引用或视图名


DELETE语句
DELETE FROM
表名
WHERE
查找条件
* 在有些数据库中，可使用可更新的表引用或视图名

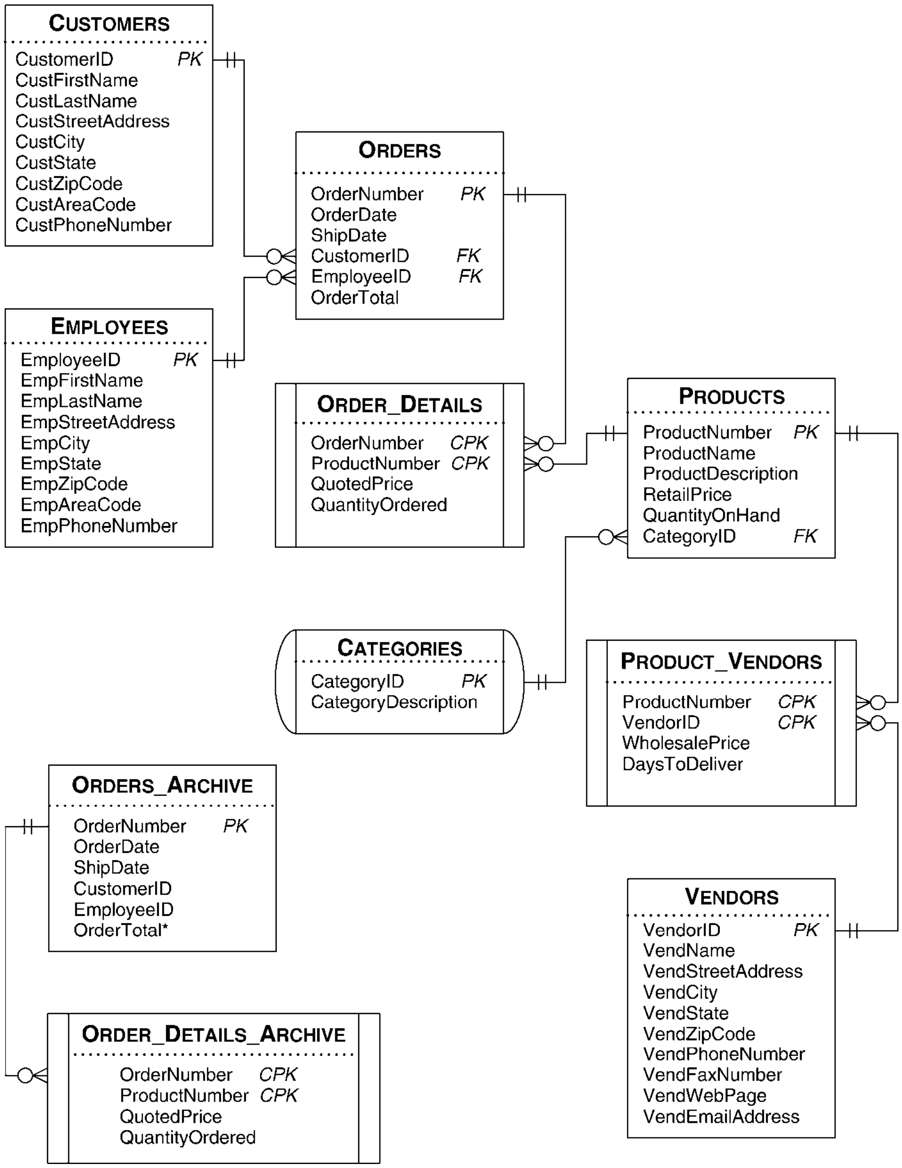
附录 B

示例数据库的结构

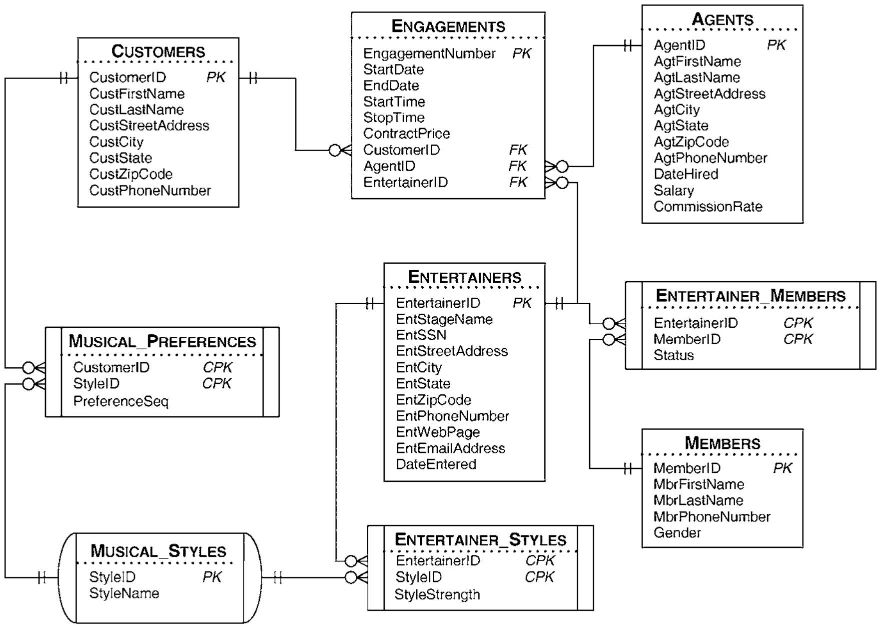
B.1 Sales Orders 示例数据库



B.2　Sales Orders Modify 数据库

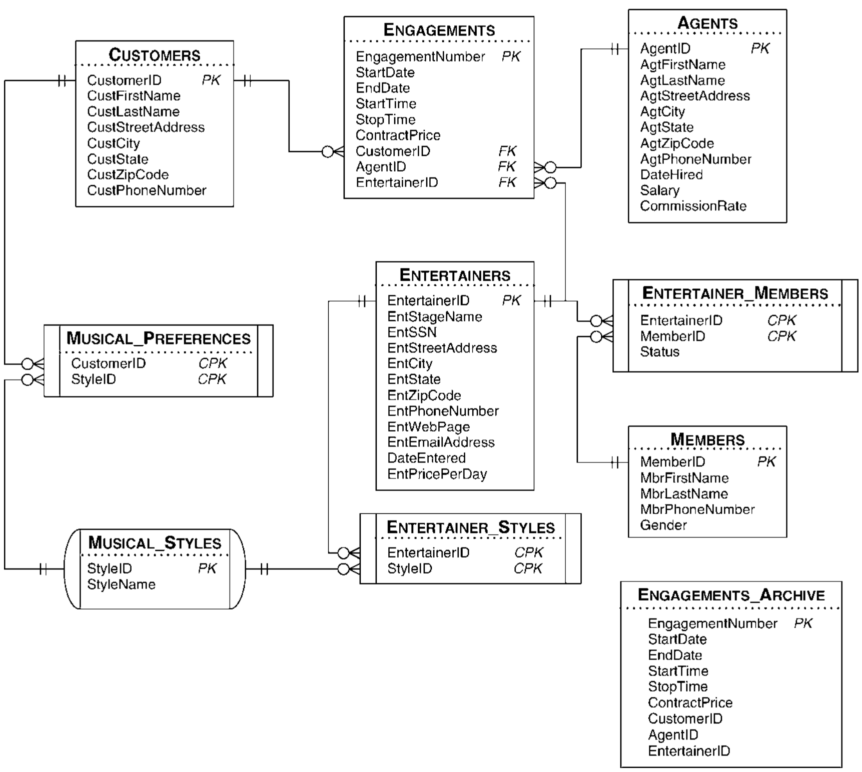


B.3　Entertainment Agency 示例数据库

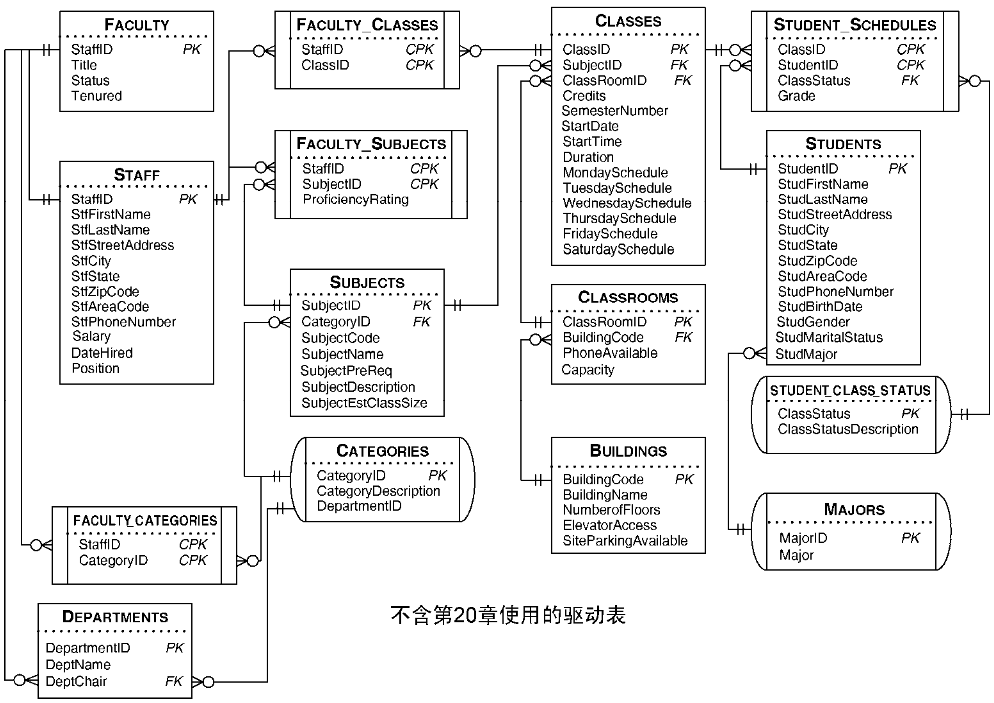


不含第20章使用的驱动表

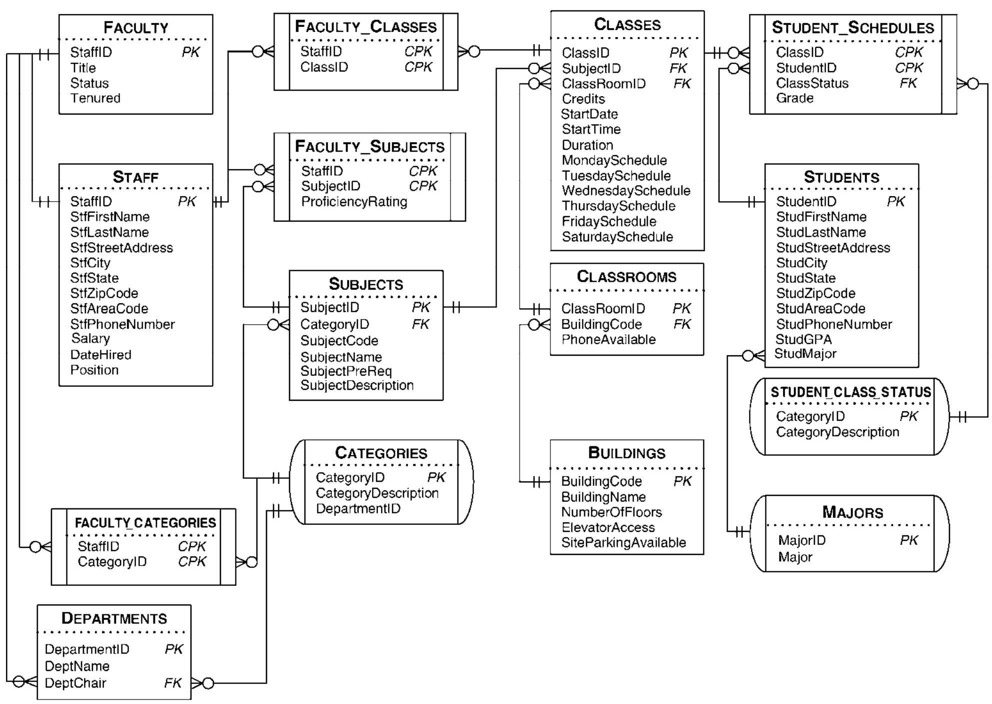
B.4　Entertainment Agency Modify 数据库



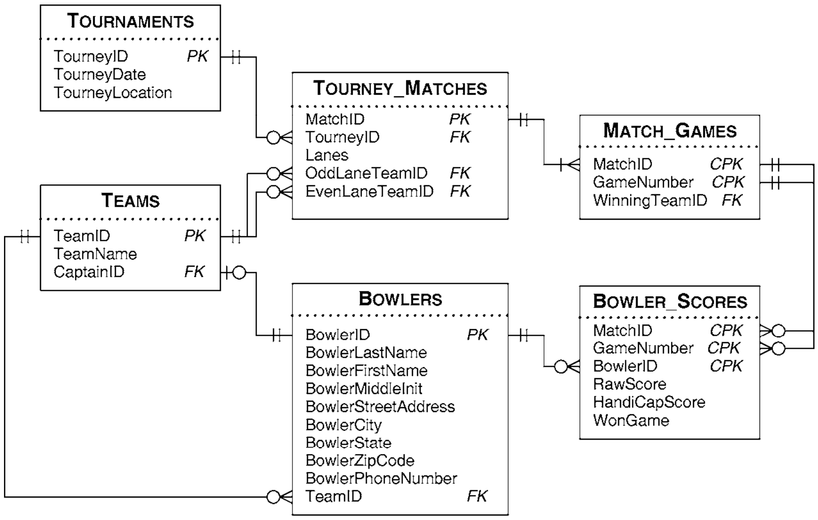
B.5 School Scheduling 示例数据库



B.6 School Scheduling Modify 数据库

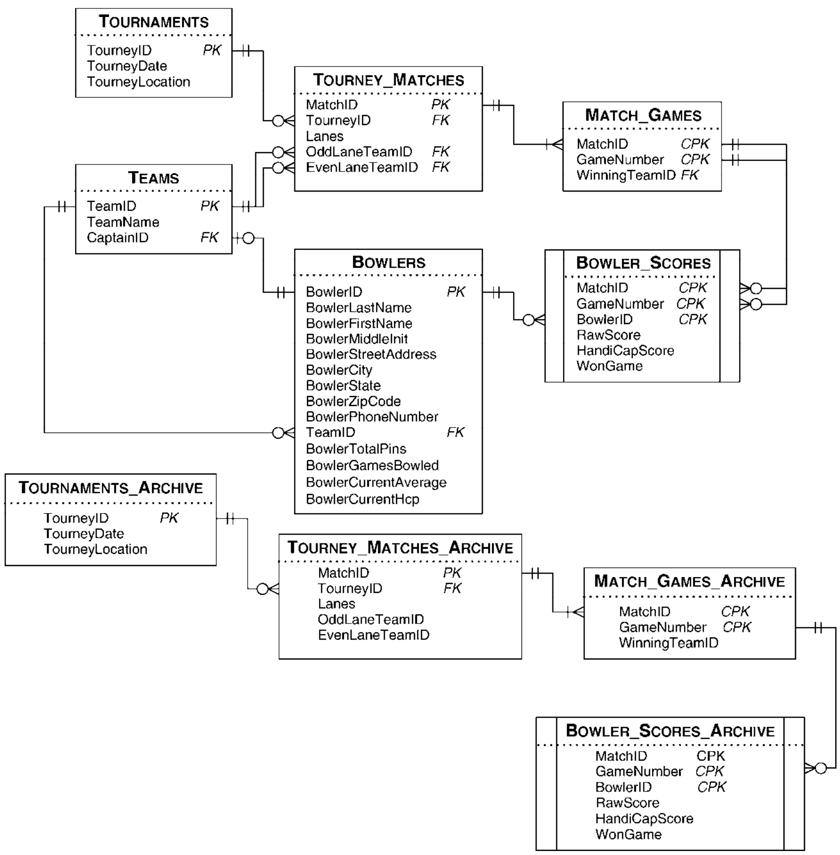


B.7　Bowling League 示例数据库

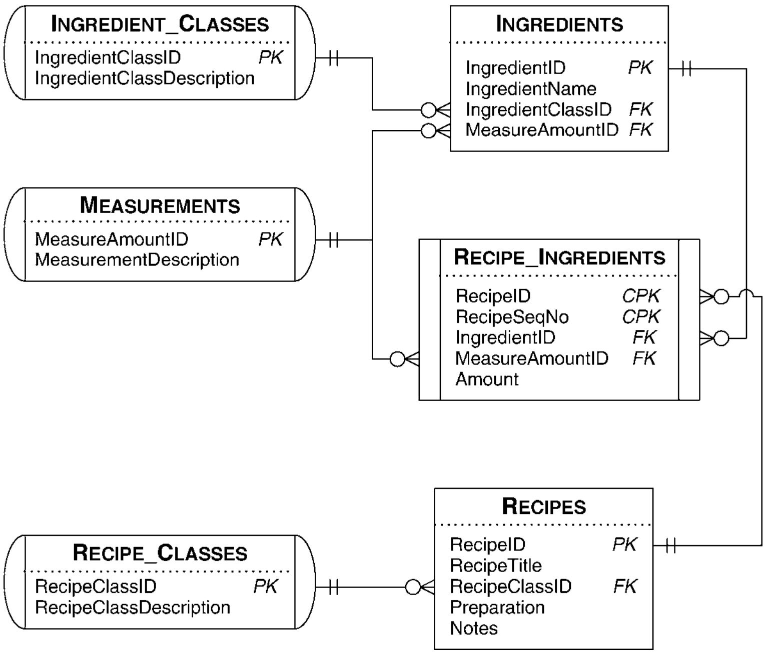


不含第20章使用的驱动表

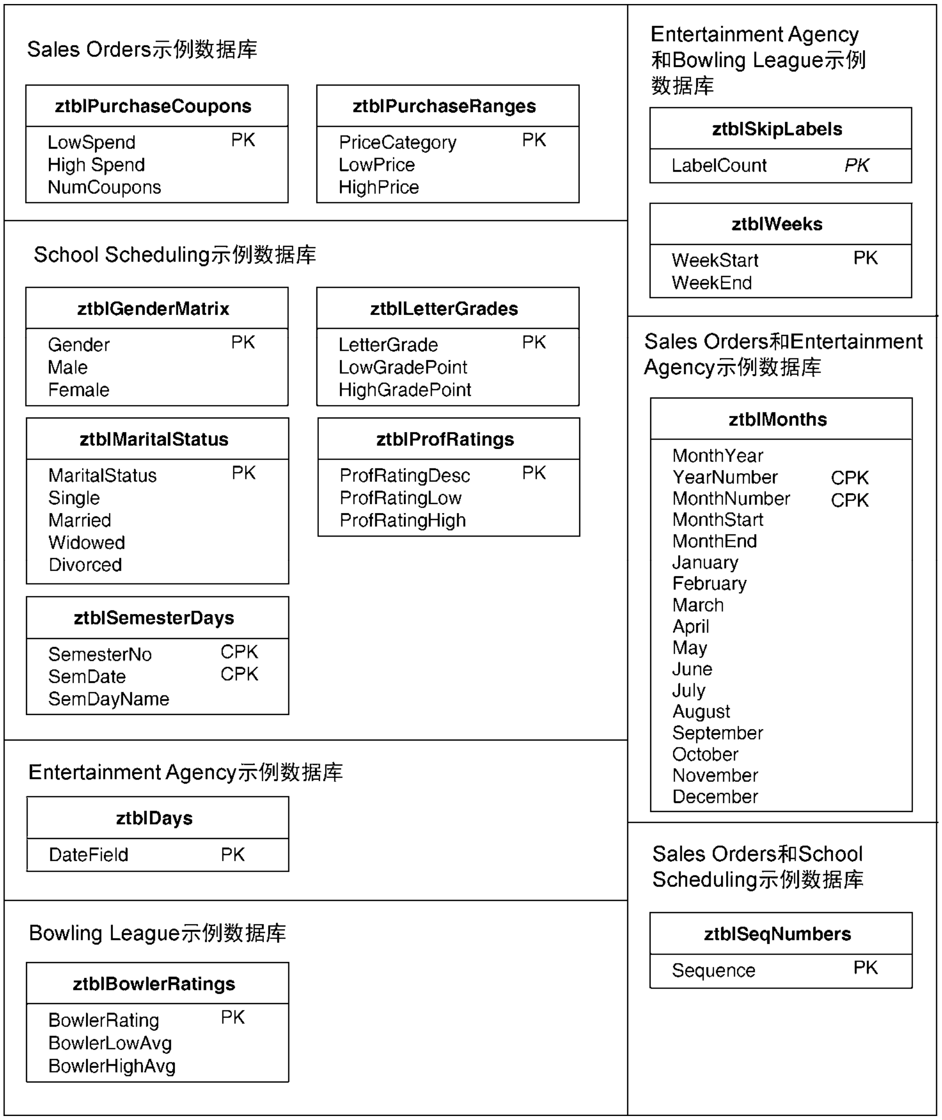
B.8　Bowling League Modify 数据库



B.9　Recipes 数据库



B.10 驱动表


Sales Orders示例数据库
ztblPurchaseCoupons
LowSpend PK
High Spend
NumCoupons
ztblPurchaseRanges
PriceCategory PK
LowPrice
HighPrice
School Scheduling示例数据库
ztblGenderMatrix
Gender PK
Male
Female
ztblLetterGrades
LetterGrade PK
LowGradePoint
HighGradePoint
ztblMaritalStatus
MaritalStatus PK
Single
Married
Widowed
Divorced
ztblProfRatings
ProfRatingDesc PK
ProfRatingLow
ProfRatingHigh
ztblSemesterDays
SemesterNo CPK
SemDate CPK
SemDayName
Entertainment Agency示例数据库
ztblDays
DateField PK
Bowling League示例数据库
ztblBowlerRatings
BowlerRating PK
BowlerLowAvg
BowlerHighAvg
Entertainment Agency和Bowling League示例数据库
ztblSkipLabels
LabelCount PK
ztblWeeks
WeekStart PK
WeekEnd
Sales Orders和Entertainment Agency示例数据库
ztblMonths
MonthYear
YearNumber CPK
MonthNumber CPK
MonthStart
MonthEnd
January
February
March
April
May
June
July
August
September
October
November
December
Sales Orders和School Scheduling示例数据库
ztblSeqNumbers
Sequence PK

附录 C

与日期和时间相关的数据类型、运算和函数

第 5 章说过，每个数据库系统都提供了很多不同的函数，可用来获取或操作日期和时间值。在日期和时间数据类型和算术方面，每个数据库系统都有其独特的规则。SQL 标准定义了 3 个函数——CURRENT_DATE、CURRENT_TIME 和 CURRENT_TIMESTAMP，但并非所有的商业数据库系统都支持它们。为了帮助你在自己使用的数据库系统中处理日期和时间值，这里概述了 6 种主要的数据库系统支持的日期和时间数据类型和算术运算，以及它们提供的日期和时间值处理函数。介绍这些函数时，指出了其名称并简要地描述了其用途。有关这些函数的使用语法，请参阅你使用的数据库系统的文档。

C.1 IBM DB2

- **支持的日期和时间数据类型**

  DATE
  TIME
  TIMESTAMP

- **支持的算术运算**

  DATE + <年数、月数、天数或日期间隔> = DATE
  DATE +/– TIME = TIMESTAMP
  TIME + <小时数、分钟数、秒数或时间间隔> = TIME
  TIMESTAMP + <日期间隔、时间间隔或日期间隔和时间间隔> = TIMESTAMP
  DATE – DATE = 日期间隔（包含 yyyymmdd 的 DECIMAL(8,0)值）
  DATE – <年数、月数、天数或日期间隔> = DATE
  TIME – TIME = 时间间隔（包含 hhmmss 的 DECIMAL(6,0)值）
  TIME – <小时数、分钟数、秒数或时间间隔> = TIME
  TIMESTAMP – TIMESTAMP = 时间间隔（包含 yyyymmddhhmmss.ms 的 DECIMAL(20,6)值）
  TIMESTAMP – <日期间隔、时间间隔或日期间隔和时间间隔> = TIMESTAMP

支持的函数如下所示。

函　数　名	描　　述
ADD_MONTHS (<*expression*>, <*number*>)	给日期或时间戳值加上指定的月数
CURDATE	获取当前日期
CURRENT_DATE	获取当前日期
CURRENT_TIME	获取本地时区的当前时间
CURRENT_TIMESTAMP	获取本地时区的当前日期和时间

（续）

函　数　名	描　　述
CURTIME	获取本地时区的当前时间
DATE(<*expression*>)	将表达式转换为日期
DAY(<*expression*>)	返回表达式表示的日期、时间戳或间隔的日部分
DAYNAME(<*expression*>)	返回表达式表示的日期、时间戳或间隔的星期几部分
DAYOFMONTH (<*expression*>)	返回表达式表示的日期或时间戳的日部分（1~31）
DAYOFWEEK (<*expression*>)	返回表达式的星期几部分，用一个数字表示（1 表示星期日）
DAYOFWEEK_ISO (<*expression*>)	返回表达式的星期几部分，用一个数字表示（1 表示星期一）
DAYOFYEAR (<*expression*>)	返回表达式表示的日期是一年中的哪一天（用 1~366 的数字表示）
DAYS(<*expression*>)	返回表达式表示的日期与 0001 年 1 月 1 日相隔的天数加 1
HOUR(<*expression*>)	返回表达式表示的时间或时间戳的小时部分
JULIAN_DAY (<*expression*>)	返回表达式表示的日期与公元前 4713 年 1 月 1 日相隔多少天
LAST_DAY(<*expression*>)	返回表达式表示的日期所在月份的最后一天
MICROSECOND (<*expression*>)	返回表达式表示的时间戳或间隔的微秒部分
MIDNIGHT_SECONDS (<*expression*>)	返回从午夜到表达式表示的时间或时间戳有多少秒
MINUTE (<*expression*>)	返回表达式表示的时间、时间戳或时间间隔的分钟部分
MONTH (<*expression*>)	返回表达式表示的日期、时间戳或日期间隔的月份部分
MONTHNAME (<*expression*>)	返回表达式表示的日期、时间戳或日期间隔的月份部分的名称
MONTHS_BETWEEN (<*expression1*>, <*expression2*>)	返回两个表达式表示的日期或时间戳大约相隔多少个月；如果 expression1 比 expression2 晚，返回的值将为正数
NEXT_DAY (<*expression*>, <*dayname*>)	返回一个时间戳，该时间戳表示的是比<expression>表示的日期晚的第一个星期几，具体是星期几由<dayname>（包含诸如 MON、TUE 等的字符串）指定
NOW	获取本地时区的当前日期和时间
QUARTER (<*expression*>)	返回表达式表示的日期属于一年的哪个季度
ROUND_TIMESTAMP (<*expression*>, <*format string*>)	将表达式舍入为与之最接近的间隔，而间隔是由格式字符串指定的
SECOND(<*expression*>)	返回表达式表示的时间、时间戳、时间间隔的秒数部分
TIME(<*expression*>)	返回表达式表示的时间或时间戳的时间部分
TIMESTAMP (<*expression1*>, [<*expression2*>])	将 expression1 表示的日期或日期时间和 expression2 表示的时间转换为一个时间戳
TIMESTAMP_FORMAT (<*expression1*>, <*expression2*>)	根据 expression2 指定的格式字符串将 expression1 指定的字符串转换为时间戳
TIMESTAMP_ISO (<*expression*>)	根据表达式表示的日期、时间或时间戳返回一个时间戳。如果表达式只包含日期，返回的时间戳将包含该日期且时间部分为零；如果表达式只包含时间，返回的时间戳将由当前日期和指定的时间组成
TIMESTAMPDIFF (<*numeric expression*>, <*string expression*>)	<numeric expression>是一个表示间隔的数字编码：1 表示毫秒；2 表示秒；4 表示分钟；8 表示小时；16 表示天；32 表示周；64 表示月；128 表示季度；256 表示年。<string expression>是将两个时间戳相减并转换为字符串的结果。这个函数返回<string expression>大约等于多少个<numeric expression>指定的间隔
TRUNC_TIMESTAMP (<*expression*>, <*format string*>)	将表达式截断为与之最接近的间隔，而间隔是由格式字符串指定的
WEEK(<*expression*>)	返回表达式的日期部分为一年的第几周：假定每年第一周都始于 1 月 1 日
WEEK_ISO(<*expression*>)	返回表达式的日期部分为该年的第几周：假定一年的第一周为该年第一个星期四所在的周
YEAR(<*expression*>)	返回表达式表示的日期或时间戳的年份部分

C.2　Microsoft Access

- **支持的日期和时间数据类型**

  Date/Time
- **支持的算术运算**

  Date/Time + Date/Time = Date/Time（结果是将两个值表示的天数的整数部分和小数部分相加得到的；每个值表示的天数是以 1899 年 12 月 31 日为第零天计算得到的）

  Date/Time − Date/Time = 两个值相隔的天数（可能包含小数部分）

  Date/Time +/− integer = Date/Time（加上或减去 integer 天）

  Date/Time +/− fraction = Date/Time（加上或减去 fraction 表示的时间——0.5 相当于 12 小时）

  Date/Time +/− integer.fraction = Date/Time

支持的函数如下所示。

函　数　名	描　　述
CDate(<*expression*>)	将表达式转换为日期值
Date	获取当前日期
DateAdd(<*interval*>, <*number*>, <*expression*>)	将 Date/Time 表达式加上指定数量的间隔
DateDiff(<*interval*>, <*expression1*>, <*expression2*>, <*firstdayofweek*>, <*firstdayofyear*>)	返回 expression1 和 expression2 表示的 Date/Time 之间相隔多少个指定的间隔。一周的第一天默认为星期日，但可指定为其他值；一年的第一周默认从 1 月 1 日开始，但可将其指定为第一个至少包含 4 天的周或第一个包含整周的周
DatePart(<*interval*>, <*expression*>, <*firstdayofweek*>, <*firstdayofyear*>)	从<expression>表示的日期或时间中提取<interval>指定的部分。一周的第一天默认为星期日，但可指定为其他值；一年的第一周默认从 1 月 1 日开始，但可将其指定为第一个至少包含 4 天的周或第一个包含整周的周
DateSerial(<*year*>, <*month*>, <*day*>)	返回与指定年份、月份和日对应的日期值
DateValue(<*expression*>)	将表达式转换为 Date/Time 值
Day(<*expression*>)	返回表达式表示的 Date/Time 值的日部分
Hour(<*expression*>)	返回表达式表示的时间的小时部分
IsDate(<*expression*>)	检查表达式表示的是否是有效的日期值，如果是就返回 True
Minute(<*expression*>)	返回表达式表示的时间的分钟部分
Month(<*expression*>)	返回表达式表示的日期的月份部分
MonthName(<*expression*>, <*abbreviate*>)	计算表达式（结果必须是 1~12 的整数）并返回相应的月份名；如果参数<abbreviate>为 True，将返回简写的月份名
Now	获取本地时区的当前日期和时间
Second(<*expression*>)	返回表达式表示的时间的秒数部分
Time	获取本地时区的当前时间
TimeSerial(<*hour*>, <*minute*>, <*second*>)	返回与指定的小时、分钟和秒对应的时间值
TimeValue（<*expression*>)	返回表达式中的时间部分
WeekDay(<*expression*>, <*firstdayofweek*>)	返回一个整数，指出表达式表示的日期是星期几；一周的第一天默认为星期日，但可将其指定为其他值
WeekDayName（<*daynumber*>, <*abbreviate*>, <*firstdayofweek*>)	返回指定的数字对应于星期几。一周的第一天默认为星期日，但可将其指定为其他值；还可要求返回星期几的缩写
Year(<*expression*>)	返回表达式表示的日期的年份部分

C.3　Microsoft SQL Server

- **支持的日期和时间数据类型**

  date
  time
  smalldatetime
  datetime
  datetime2
  datetimeoffset
- **支持的算术运算**

  datetime + datetime = datetime（结果是将两个值表示的天数的整数部分和小数部分相加得到的；每个值表示的天数是以 1899 年 12 月 31 日为第零天计算得到的）
  datetime +/− integer = datetime（加上或减去 integer 天）
  datetime +/− fraction = datetime（加上或减去 fraction 表示的时间——0.5 相当于 12 小时）
  datetime +/− integer.fraction = datetime
  datetime − datetime = 两个值相隔的天数（可能包含小数部分）
  smalldatetime + smalldatetime = smalldatetime（结果是将两个值表示的天数的整数部分和小数部分相加得到的；每个值表示的天数是以 1899 年 12 月 31 日为第零天计算得到的）
  smalldatetime +/− integer = smalldatetime（加上或减去 integer 天）
  smalldatetime +/− fraction = smalldatetime（加上或减去 fraction 表示的时间——0.5 相当于 12 小时）
  smalldatetime + integer.fraction = smalldatetime
  smalldatetime − smalldatetime =两个值相隔的天数（可能包含小数部分）

支持的函数如下所示。

函　数　名	描　　述
CURRENT_TIMESTAMP	获取本地时区的当前日期和时间
DATEADD(<*interval*>, <*number*>, <*expression*>)	将日期/时间表达式加上指定数量的间隔
DATEDIFF (<*interval*>, <*expression1*>, <*expression2*>)	返回 expression1 和 expression2 表示的日期时间之间相隔多少个指定的间隔
DATEFROMPARTS(<*year*>, <*month*>, <*day*>)	返回与指定年份、月份和日对应的日期值
DATENAME(<*interval*>, <*expression*>)	返回表达式表示的日期的指定部分；如果<interval>指定的是月份或星期几，将返回完整的名称
DATEPART(<*interval*>, <*expression*>)	返回表达式表示的日期或时间的指定部分，并用整数表示
DATETIMEFROMPARTS (<*year*>, <*month*>, <*day*>, <*hour*>, <*minute*>, <*second*>, <*millisecond*>)	返回指定年份、月份、日、小时、分钟、秒和毫秒对应的日期时间值
DATETIME2FROMPARTS (<*year*>, <*month*>, <*day*>, <*hour*>, <*minute*>, <*second*>, <*fraction*>, <*precision*>)	返回指定年份、月份、日、小时、分钟、秒、小数部分和精度对应的 datetime2 值
DAY(<*expression*>)	返回表达式表示的日期的日部分
EOMONTH(<*date*> [,<*months*>])	返回指定日期所属月份的最后一天；参数<months>是可选的，如果指定了它，将把指定日期加上指定的月数，再返回得到的日期所属月份的最后一天
GETDATE()	返回一个表示当前日期的 datetime 值
GETUTCDATE()	返回一个表示当前 UTC（Coordinated Universal Time，世界统一时间）日期的 datetime 值
ISDATE(<*expression*>)	检查表达式表示的是否是有效的日期值，如果是就返回 1

（续）

函　数　名	描　　述
MONTH(<*expression*>)	返回一个整数，指出表达式表示的日期值的月份部分
SMALLDATETIMEFROMPARTS(<*year*>, <*month*>, <*day*>, <*hour*>, <*minute*>, <*second*>)	返回指定年份、月份、日、小时、分钟和秒对应的 smalldatetime 值
SWITCHOFFSET (<*datetimeoffset*>, <*offset*>)	将指定 datetimeoffset 值的时区偏移量改为指定的值，并返回得到的 datetimeoffset 值
SYSDATETIME()	返回一个表示当前日期和时间的 datetime2 值
SYSDATETIMEOFFSET()	返回一个表示当前日期和时间（包括时区偏移量）的 datetimeoffset 值
SYSUTCDATETIME()	返回一个表示当前 UTC 日期和时间的 datetime2 值
TIMEFROMPARTS(<*hour*>, <*minute*>, <*second*>, <*fraction*>, <*precision*>)	返回指定小时、分钟、秒、小数部分和精度对应的时间值
TODATETIMEOFFSET (<*datetime2*>, <*offset*>)	根据指定的时区偏移量将一个 datetime2 值转换为一个 datetimeoffset 值
YEAR(<*expression*>)	返回一个整数，指出表达式表示的日期的年份部分

C.4　MySQL

- **支持的日期和时间数据类型**

  date

  datetime

  timestamp

  time

  year

- **支持的算术运算**

  date +/– Interval <interval [year/quarter/month/week/day]> = date

  datetime +/– Interval <interval [year/quarter/month/week/day/hour/minute/second]> = datetime

  timestamp +/– Interval <interval [year/quarter/month/week/day/ hour/minute/second]> = timestamp

  time +/– Interval <interval [hour/minute/second]> = time

❖ **说明**　对于数据类型 date 和 time，也可将其与整数或小数相加减，但 MySQL 将先把 date 或 time 值转换为数字，再执行加减运算。例如，将 date 值“2017-11-15”与 30 相加时，结果为 20171145；将 time 值“12:20:00”与 100 相加时，结果为 122100。执行 date 和 time 算术运算时，务必使用关键字 INTERVAL。

支持的函数如下所示。

函　数　名	描　　述
ADDDATE(<*expression*>, <*days*>)	将表达式表示的 date 值加上指定的天数
ADDDATE(<*expression*>, INTERVAL <*amount*> <*units*>)	将表达式表示的 date 值加上指定数量的指定间隔
ADDTIME(<*expression*>, <*time*>)	将表达式表示的 time 或 datetime 值加上指定的时间
CONVERT_TZ(<*datetime*>, <*from tz*>, <*to tz*>)	将 datetime 值从一个时区转换为另一个时区
CURRENT_DATE、CURDATE()	获取当前日期
CURRENT_TIME、CURTIME()	获取本地时区的当前时间
CURRENT_TIMESTAMP	获取本地时区的当前日期和时间
DATE(<*expression*>)	从表达式表示的 datetime 值中提取日期部分

（续）

函　数　名	描　　述
DATE_ADD(*<expression>*, INTERVAL *<interval>* *<quantity>*)	将表达式表示的 date 或 datetime 值加上指定数量的指定间隔
DATE_SUB(*<expression>*, INTERVAL *<interval>* *<quantity>*)	将表达式表示的 date 或 datetime 值减去指定数量的指定间隔
DATEDIFF(*<expression1>*, *<expression2>*)	将两个表达式表示的 datetime 值相减，并返回它们之间相隔的天数
DAY(*<expression>*)	返回表达式表示的 date 值的日部分，并用 1~31 的数字表示
DAYNAME(*<expression>*)	返回表达式表示的 date 或 datetime 值为星期几，并用名称表示
DAYOFMONTH (*<expression>*)	返回表达式表示的 date 值的日部分，并用 1~31 的数字表示
DAYOFWEEK(*<expression>*)	返回表达式表示的 date 或 datetime 值为星期几，并用数字表示（1 表示星期日）
DAYOFYEAR(*<expression>*)	返回一个 1~366 的数字，指出表达式表示的日期是当年的哪一天
EXTRACT(*<unit>* FROM *<expression>*)	返回表达式表示的日期时间的指定部分，如年份或月份
FROM_DAYS(*<number>*)	返回比公元前 1 年 12 月 31 日晚指定天数的日期；例如，晚 366 天的日期为公元 1 年 1 月 1 日
HOUR(*<expression>*)	返回表达式表示的时间的小时部分
LAST_DAY(*<expression>*)	返回表达式表示的日期所属月份的最后一天
LOCALTIME 和 LOCALTIMESTAMP	参见函数 NOW
MAKEDATE(*<year>*, *<dayofyear>*)	返回指定年份和该年的第多少天（1~366）对应的日期
MAKETIME(*<hour>*, *<minute>*, *<second>*)	返回指定小时、分钟和秒对应的时间
MICROSECOND (*<expression>*)	返回表达式表示的时间或日期时间值中的毫秒部分
MINUTE(*<expression>*)	返回表达式表示的时间或日期时间值中的分钟部分
MONTH(*<expression>*)	返回表达式表示的日期值的月份部分
MONTHNAME (*<expression>*)	返回表达式表示的时间或日期时间值中的月份部分，用月份名表示
NOW()	获取本地时区的当前日期和时间
QUARTER(*<expression>*)	返回表达式表示的日期属于哪个季度，用数字表示
SECOND(*<expression>*)	返回表达式表示的时间或日期时间值的秒部分
STR_TO_DATE (*<expression>*, *<format>*)	根据指定的格式将表达式转换为一个日期值、日期时间值或日期值
SUBDATE(*<expression>*, INTERVAL *<interval>* *<quantity>*)	参见函数 DATE_SUB
SUBTIME(*<expression1>*, *<expression2>*)	将 expression1 表示的日期时间或时间减去 expression2 表示的时间，并返回得到的时间值或日期时间值
TIME(*<expression>*)	返回表达式表示的时间值或日期时间值的时间部分
TIME_TO_SEC (*<expression>*)	将表达式表示的时间转换为秒数
TIMEDIFF(*<expression1>*, *<expression2>*)	将 expression1 和 expression2 表示的时间或日期时间值相减，并返回结果
TIMESTAMP(*<expression>*)	返回表达式表示的日期时间值
TIMESTAMP(*<expression1>*, *<expression2>*)	将 expression1 表示的日期或日期时间值加上 expression2 表示的时间值，并返回得到的日期时间值
TIMESTAMPADD (*<interval>*, *<number>*, *<expression>*)	将表达式表示的日期或日期时间值加上指定数量的指定间隔
TIMESTAMPDIFF (*<interval>*, *<expression1>*, *<expression2>*)	返回一个数字，指出 expression1 和 expression2 表示的日期或日期时间值相隔多少个指定的间隔
TO_DAYS(*<expression>*)	返回从公元 0 年到表达式表示的日期有多少天
UTC_DATE	获取当前的 UTC 日期
UTC_TIME	获取当前的 UTC 时间

（续）

函　数　名	描　　述
UTC_TIMESTAMP	获取当前的 UTC 日期和时间
WEEK(<*expression*>, <*mode*>)	根据指定的模式返回表达式表示的日期属于当年的第几周
WEEKDAY(<*expression*>)	返回一个整数，指出表达式表示的日期是星期几（0 表示星期一）
WEEKOFYEAR (<*expression*>)	返回表达式表示的日期属于当年的第几（1~53）周，假定第一周至少有 4 天
YEAR(<*expression*>)	返回表达式表示的日期的年份部分

C.5　Oracle

- **支持的日期和时间数据类型**

  DATE
  TIMESTAMP
  INTERVAL YEAR TO MONTH
  INTERVAL DAY TO SECOND

- **支持的算术运算**

  DATE + INTERVAL = DATE
  DATE + 数字 = DATE
  DATE – DATE = 数字（单位为天，可能包含小数部分）
  DATE – TIMESTAMP = INTERVAL
  DATE – INTERVAL = DATE
  DATE – 数字 = DATE
  INTERVAL + DATE = DATE
  INTERVAL + TIMESTAMP = TIMESTAMP
  INTERVAL + INTERVAL = INTERVAL
  INTERVAL – INTERVAL = INTERVAL
  INTERVAL × 数字 = INTERVAL
  INTERVAL / 数字 = INTERVAL

支持的函数如下所示。

函　数　名	描　　述
ADD_MONTHS(<*date*>, <*integer*>)	返回指定日期加上指定月数的结果
CURRENT_DATE	获取当前日期
CURRENT_TIMESTAMP	获取本地时区的当前日期、时间和时间戳
DBTIMEZONE	获取数据库的时区
EXTRACT(<*interval*> FROM <*expression*>)	返回表达式表示的日期时间的指定部分（年份、月份、日等）
LOCALTIMESTAMP	获取本地时区的当前日期和时间
MONTHS_BETWEEN (<*expression1*>, <*expression2*>)	计算 expression2 和 expression1 相差多少个月（可能包含小数部分）
NEW_TIME(<*expression*>, <*timezone1*>, <*timezone2*>)	将表达式表示的第一个时区的日期时间转换为第二个时区的日期时间
NEXT_DAY(<*expression*>, <*dayname*>)	返回晚于表达式表示的日期的第一个星期几的日期，具体是星期几由<dayname>（一个包含诸如“MONDAY”、“TUESDAY”等的字符串）指定
NUMTODSINTERVAL (<*number*>, <*unit*>)	将指定数量的单位时间（DAY、HOUR、MINUTE、SECOND）转换为 INTERVAL DAY TO SECOND

（续）

函　数　名	描　　述
NUMTOYMINTERVAL (<*number*>, <*unit*>)	将指定数量的单位时间（YEAR、MONTH）转换为 INTERVAL YEAR TO MONTH
ROUND(<*expression*>, <*interval*>)	将指定日期值舍入到指定的间隔
SESSIONTIMEZONE	获取当前会话的时区（session）
SYSDATE	获取数据库服务器的当前日期和时间
SYSTIMESTAMP	获取数据库服务器的当前日期、时间和时区
TO_DATE(<*expression*>, <*format*>)	使用指定的格式将字符串表达式转换为一种日期数据类型
TO_DSINTERVAL (<*expression*>)	将字符串表达式转换为 INTERVAL DAY TO SECOND
TO_TIMESTAMP (<*expression*>, <*format*>)	使用指定的格式将字符串表达式转换为一种时间戳数据类型
TO_TIMESTAMP_TZ (<*expression*>, <*format*>)	使用指定的格式将字符串表达式转换为一种包含时区的时间戳数据类型
TO_YMINTERVAL	将字符串表达式转换为 INTERVAL YEAR TO MONTH
TRUNC(<*expression*>, <*interval*>)	将指定日期值截断到指定的间隔

C.6　PostgreSQL

- **支持的日期和时间数据类型**

  DATE
  TIME（包含或不包含时区）
  TIMESTAMP（包含或不包含时区）
  INTERVAL

- **支持的算术运算**

  DATE +/– INTERVAL = TIMESTAMP
  DATE +/– 数字 = DATE
  DATE + TIME = TIMESTAMP
  DATE – DATE = 整数
  TIME +/– INTERVAL = TIME
  TIME – TIME = INTERVAL
  TIMESTAMP +/– INTERVAL = TIMESTAMP
  TIMESTAMP – TIMESTAMP = INTERVAL
  INTERVAL +/– INTERVAL = INTERVAL
  INTERVAL × 数字 = INTERVAL
  INTERVAL / 数字 = INTERVAL

支持的函数如下所示。

函　数　名	描　　述
AGE(<*timestamp*>, <*timestamp*>)	将两个参数相减，得到一个使用年和月的符号化结果
AGE(<*timestamp*>)	从当前日期（时间为午夜）减去指定的时间戳
CLOCK_TIMESTAMP()	当前日期和时间（语句执行期间会变）
CURRENT_DATE	当前日期
CURRENT_TIME	当前时间
CURRENT_TIMESTAMP	当前日期和时间（当前事务开始时）

（续）

函　数　名	描　　述
DATE_PART(<*text*>, <*timestamp*>)	从<timestamp>中获取<text>指定的部分(year、month、day 等), 与 EXTRACT 等效
DATE_PART(<*text*>, <*interval*>)	从<interval>中获取<text>指定的部分(day、minute、second 等), 与 EXTRACT 等效
DATE_TRUNC(<*text*>, <*timestamp*>)	将<timestamp>截断到<text>指定的精度（microseconds、milliseconds、minute 等）
EXTRACT(<*text*> FROM <*timestamp*>)	从<timestamp>中获取<text>指定的部分（year、month、day 等）
EXTRACT(<*text*> FROM <*interval*>)	从<interval>中获取<text>指定的部分（day、minute、second 等）
ISFINITE(<*date*>)	检查指定的日期是否是有穷的（不是正/负无穷大）
ISFINITE(<*timestamp*>)	检查指定的时间戳是否是有穷的（不是正/负无穷大）
ISFINITE(<*interval*>)	检查指定的间隔是否是有穷的（不是正/负无穷大）
JUSTIFY_DAYS (<*interval*>)	按每月 30 天调整间隔
JUSTIFY_HOURS (<*interval*>)	按每天 24 小时调整间隔
JUSTIFY_INTERVAL (<*interval*>)	使用 JUSTIFY_DAYS 和 JUSTIFY_HOURS 调整间隔，并调整符号
LOCALTIME	当前时间
LOCALTIMESTAMP	当前日期和时间（当前事务开始时）
NOW()	当前日期和时间（当前事务开始时）
STATEMENT_TIMESTAMP()	当前日期和时间（当前语句开始时）
TIMEOFDAY()	当前日期和时间（类似于 CLOCK_TIMESTAMP，但返回的是一个文本字符串）
TRANSACTION_TIMESTAMP()	当前日期和时间（当前事务开始时）

附录 D 推荐读物

要更深入地了解数据库设计和 SQL，建议阅读如下图书。记住，这里列出的有些图书技术性很强，可能不容易看懂。另外，有些图书假定你掌握了计算机、数据库和编程的基础知识。

D.1 数据库图书

Connolly, Thomas, and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management* (6th ed.). Essex, England: Addison-Wesley, 2014.

Coronel, Carlos and Steven Morris. *Database Systems: Design, Implementation, and Management* (11th ed.). Stamford, CT: Cengage Learning, 2015.

Date, C.J. *An Introduction to Database Systems* (8th ed.). Boston, MA: Addison-Wesley, 2003.

Date, C.J. *Database in Depth: Relational Theory for Practitioners*. Sebastopol, CA: O'Reilly Media, 2005.

Date, C.J. *Database Design and Relational Theory: Normal Forms and All That Jazz.* Sebastopol, CA: O'Reilly Media, 2012.

Hernandez, Michael J. *Database Design for Mere Mortals* (3rd ed.). Boston, MA: Addison-Wesley, 2013.

D.2 SQL 图书

Bowman, Judith S., Sandra L. Emerson, and Marcy Darnovsky. *The Practical SQL Handbook: Using SQL Variants* (4th ed.). Boston, MA: Addison-Wesley, 2001.

Celko, Joe. *Joe Celko's SQL for Smarties: Advanced SQL Programming* (5th ed.). Burlington, MA: Morgan Kaufmann Publishers, 2014.

Date, C.J., and Hugh Darwen. *A Guide to the SQL Standard* (4th ed.). Reading, MA: Addison-Wesley, 1996.

Rockoff, Larry. *The Language of SQL* (2nd ed.). Boston, MA: Addison-Wesley, 2016.

Viescas, John L., Douglas J. Steele, and Ben G. Clothier. *Effective SQL: 61 Ways to Write Better SQL.* Boston, MA: Addison-Wesley, 2016.

结　　语

“对一向明白的事物突然有了全新的领悟，这就是学习的真谛。”

——多丽丝·莱辛

至此，你掌握了查询和修改数据库中数据所需的全部工具。你学习了如何编写或简单或复杂的 SELECT 语句来处理各种数据，如何使用查找条件筛选数据，如何通过连接使用多个表，如何通过对数据进行分组来生成统计信息，如何更新、添加和删除表中数据，如何跳出习惯性思维从而创造性地解决否定型和多条件型问题，如何使用条件测试，如何使用非连接表，如何进行复杂分组，以及如何透过“窗口”观察数据。

与所有学习过程一样，总有新的东西等着你去学习。你接下来的任务是，在数据库系统中将你通过阅读本书学到的知识付诸应用。请务必参阅你使用的数据库系统的文档，找出标准 SQL 语法与该数据库系统使用的 SQL 语法有何不同。在你使用的数据库系统中，如果可使用图形界面来创建查询，你可能会发现，这种界面理解和使用起来都容易得多。

另外，别忘了，本书只介绍了 SQL 的数据操作部分。如果你备受鼓舞，还有很多方面等待你去探索。例如，你可以学习如何创建数据结构，如何在视图、函数或存储过程中使用多个表和命令，以及如何在应用程序中嵌入 SQL 语句。要更深入地学习 SQL，建议你首先阅读附录 D 的推荐读物。

我很享受编写本书的过程，但愿你也很享受阅读本书的过程。我知道介绍 SQL 的图书通常非常乏味，因此我竭尽所能地添加了一点趣味感和幽默感。学习绝不应烦琐而乏味，相反，你应该对每天学习一点新东西充满期待。

著述是一个让人意识到不足的过程，它让你认识到还有很多需要学习的东西。在编著过程中，不可避免地要从全新的角度看待问题，我由此发现前述多丽丝·莱辛的话实乃至理名言。

但愿你也这样认为。

SQL查询
从入门到实践
（第4版）

“本书循序渐进地介绍了如何编写SQL查询，语言清晰易懂，并且通过大量的示例和详细说明来帮助你掌握理解、修改和编写SQL查询所需的技能。我可以负责任地说，作者不仅谙熟SQL，而且擅长向人讲述SQL，这两种品质让本书成了一份极其宝贵的参考资料。”

—— Keith W. Hare
JCC咨询公司高级咨询师
美国SQL标准委员会副主席
国际SQL标准委员会召集人

“如果你需要处理数据，强烈建议你阅读这本书，它将让你轻松学会数据处理的一个最重要的方面——编写查询。查询是重要的数据选择、排序和报告工具，可弥补表结构的不足、满足新的报告需求、纳入新的数据源。本书语言清晰易懂，通过大量的示例演示了如何解决从简单到复杂的问题，并将概念应用于各种不同的场景。无论你是新手还是专家，本书都极具参考价值。”

—— Teresa Hennig
Microsoft Access MVP

“除非要以非常复杂的方式使用SQL，否则阅读本书就够了。它揭开了复杂查询的神秘面纱，清晰地诠释了原理和方法，赋予普通人以超人的力量。千万不要错过！”

—— Graham Mandeno
数据库咨询师

“本书首先介绍如何设计数据库表，让SQL能够发挥作用（这是初学者都会遇到的问题），接着全面讨论了SQL的构成和功能。读完本书并完成其中的示例后，你就是半个专家了。通过学会将问题转换为SQL语句，你的SQL水平将得到极大的提高。书中有大量实例帮助你明白如何使用SQL来解决有关数据库的问题。只要有一个提示能让你在编写查询时得以避开陷阱，本书就物超所值了。要想充分发挥数据库的潜力，强烈建议你阅读这本书。”

—— Kenneth D. Snell博士
数据库设计师

“真是精妙绝伦！长期以来，我一直在使用SQL，但本书介绍的方法让我在学习过程中养成的一些坏习惯暴露无遗。真希望我很久以前就学会了这些方法，这样我的SQL学习之路就不会那么艰难曲折了。”

—— Leo（theDBguy）
Utter Access管理员、Microsoft Access MVP

Pearson
www.pearson.com

图灵社区：iTuring.cn
热线：(010)51095183转600

分类建议 计算机 / 数据库 / SQL

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-53401-9
9 787115 534019 >

ISBN 978-7-115-53401-9
定价：149.00元